Dependent Stringly-Typed Programming

gallais

March 26, 2021

1 Introduction

Static type systems started as a lightweight compile time check enforcing basic hygiene by preventing users from e.g. adding an integer to a boolean. Seduced by the promise of ever more guarantees computer scientists have invented ever more powerful type systems to the point where they could formalise all of mathematics as types and programs.

In this paper we reclaim this power to the advantage of the honest working programmer by demonstrating how one can use the ivory tower concepts to revitalise the age old practice of stringly typed programming. We will use Agda [Norell(2009)] as our language of choice because it provides us with powerful enough "unsafe" primitives to conduct our experiment.

This paper is a self-contained literate Agda file so the interested reader should be able to independently reproduce our results. You can also find the content on github at `https://github.com/gallais/STRINaGda`.

2 What even is a type?

For our purposes, we will simply say that a type is a function that, given a linked list of characters, tells us whether we should accept it or not as a value of that type. Luckily Agda provides us with builtin notions of List, Char, and Bool so this is easily defined.

```
open import Agda.Builtin.List   using (List)
open import Agda.Builtin.Char using (Char)
open import Agda.Builtin.Bool  using (Bool)

Type = List Char → Bool
```

Next we can define what it means to belong to a type. By definition, a list of characters belongs to a type if the function returns the boolean true when run on that list. To make this formal we need to define an Agda function internalising the predicate "this boolean is true".

Agda ships with a notion of trivial truthfulness (the unit type) but unfortunately it does not provide us with a notion of trivial falsity. So we have to define the empty type ourselves as a sum type with zero constructor.

```
open import Agda.Builtin.Unit using (⊤)

data ⊥ : Set where
```

Equipped with trivial truthfulness and trivial falsity, we can internalise what it means for a boolean to be true by pattern matching on it and returning the unit type if it is true, or the empty one if it is false.

```
open Agda.Builtin.Bool using (true; false)

IsTrue : Bool → Set
IsTrue true  = ⊤
IsTrue false = ⊥
```

This is precisely what we need to express what it means for a list of characters to belong to a given type: run the type on the list of characters and check it returned true.

```
_∈_ : List Char → Type → Set
cs ∈ A = IsTrue (A cs)
```

We can define a convenient wrapper for elements of a given type by packing a list of characters together with the proof that it is accepted at that type. We use a dependent record and make the check field an erased instance argument, that is to say that we never want to have to write these proofs explicitly, expect Agda to just automatically pass them around without needing our help, and to forget about them when compiling the code.

```
record Elt (A : Type) : Set where
  constructor [_]
  field       value : List Char
  field @0 {{check}} : value ∈ A
open Elt
```

Agda's string literals are tied to its builtin notion of String which is distinct from List Char. We can luckily convert from one to the other by unpacking the string. We define a convenient helper function to, given a string and a type, return an element of that type by checking that the unpacked string is accepted at that type. This will help us write concrete examples and unit tests.

```
open import Agda.Builtin.String using (String)
  renaming (primStringToList to unpack)

infix 100 _∋_
_∋_ : (A : Type) (str : String) →
      {{unpack str ∈ A}} → Elt A
A ∋ str = record { value = unpack str }
```

We now have a formal definition of what a type is, what it means for a string to be accepted at a given type and what an element of a type looks like. Let us look at a concrete example of a type.

3 Our First Type: ℕ

As is customary in any document talking about dependent types, we will start by defining the natural numbers. The customary presentation is that a natural number is either zero or the successor of a natural natural number. In terms of strings, we will characterise this as being either the "Z" string or a string starting with 'S' and whose tail is itself a natural number.

Agda, being a very inpractical programming language, does not ship with _&&_ and _||_ defined on booleans. The standard library does provide these definitions but has to be installed separately and we want this document to be self-contained so we will have to start by defining them ourselves.

```
infixr 3 _&&_
_&&_ : Bool → Bool → Bool
true  && b = b
false && b = false

infixr 2 _||_
_||_ : Bool → Bool → Bool
true  || b = true
false || b = b
```

Next we need a way to test that a list of characters is empty. The builtin type List has two constructors: [] for the empty list, and _::_ for putting together a character as the head of the linked list and a tail of additional characters. A list is empty precisely when it is [].

```
open Agda.Builtin.List using ([]; _::_)

isNil : List Char → Bool
isNil []       = true
isNil (_ :: _) = false
```

The last piece of the puzzle is the ability to test two characters for equality. This is once again provided as a primitive by Agda and we import it and simply rename it to make the code more readable.

```
open Agda.Builtin.Char
  renaming (primCharEquality to _==_)
```

We are now ready to define the type of natural numbers. A beautiful thing about stringly typed programming is that we can assign a very precise type to each constructor of a datatype. So we not only define the type ℕ but also mutually introduce the types isZ and isS of the zero and successor constructors respectively.

```
ℕ   : Type
isZ : Type
isS : Type
```

The type of natural numbers is exactly the union of the type of zero and successors.

```
ℕ cs = isZ cs || isS cs
```

The types of zero and successor are defined by case analysis on the input list of characters. If the list is empty then it does not belong to any of these types. If it is non-empty then we check that it is either 'Z'-headed and with an empty tail for the zero type, or 'S'-headed and with a tail that is itself a natural number in the successor case.

```
isZ [] = false
isZ (c :: cs) = c == 'Z' && isNil cs

isS [] = false
isS (c :: cs) = c == 'S' && ℕ cs
```

Unsurprisingly we can define the zero and suc constructors for ℕ. Note that we do not need to write any proofs that the strings are valid: Agda takes care of the proofs for us by a mix of computation and implicit proof construction.

```
zero : Elt ℕ
zero = ℕ ∋ "Z"

suc : Elt ℕ → Elt ℕ
suc [ n ] = [ 'S' :: n ]
```

We can define constant numbers either by using our _∋_ gadget or by using suc and zero, whatever feels most convenient.

```
one   = ℕ ∋ "SZ"
two   = suc (suc zero)
three = ℕ ∋ "SSSZ"
four  = suc three
```

We will use these constants again when writing unit tests for the programs over natural numbers we are now going to develop.

Now that we have our notion of types, a working example and even some inhabitants, it would be nice to be able to do something with them.

4 Stringly Typed Programming

Being able to construct values of a given type is all well and good but we, as programmers, want to be able to take them apart too.

Induction is the dependently typed generalisation of primitive recursion: for a predicate P on values of type ℕ, if we can prove that P zero holds and that for any natural number n, if $P\ n$ holds then so does P (suc n) then we ought to be able to have a function computing from a natural number n a proof of type $P\ n$.

4.1 Small Scale Reflection

The tricky part in defining induction for the natural numbers is in connecting the observations made by the builtin boolean-valued equality test on characters _==_ with propositional equality.

We introduce a Reflects inductive family [Dybjer(1994)] indexed by two Chars and a Bool. Inspired by the architecture of Coq's small scale reflection library [Mahboubi and Tassi(2021)], it formalises the fact that whenever the boolean is true then the two characters are equal.

We name the Reflects constructors the same as the boolean constructor they are respectively indexed by. This means that matching on such a proof looks like matching on the original boolean.

```
data Reflects (c : Char) : Char → Bool → Set where
  true : Reflects c c true
  false : ∀ {d} → Reflects c d false
```

We can readily prove that if a and b are known to be the same according to Agda's builtin notion of propositional equality then we have that Reflects a b true.

```
open import Agda.Builtin.Equality using (_≡_; refl)
```

```
mkTrue : ∀ {a b} → a ≡ b → Reflects a b true
mkTrue refl = true
```

The only thing missing for us is a proof that whenever the boolean test a == b returns true then the values are indeed propositionally equal i.e. $a \equiv b$. Unfortunately Agda does not provide a primitive proof of this fact. We will have to use an unsafe primitive called trustMe to build such a proof.

```
open import Agda.Builtin.TrustMe
  renaming (primTrustMe to trustMe)
```

By combining mkTrue and trustMe we can write a function demonstrating that the (a == b) test produces a boolean that reflects a test on propositional equality.

```
_=?_ : (a b : Char) → Reflects a b (a == b)
a =? b with a == b
... | false = false
... | true = mkTrue trustMe
```

4.2 Induction principle for ℕ

And with that in our backpocket we are well equipped to prove induction. First we use an anonymous module to parametrise all of the following definitions over the same predicate P, proof of the base case $P0$ and proof of the step case PS.

```
module _ (P : Elt ℕ → Set)
         (P0 : P zero)
         (PS : ∀ n → P n → P (suc n))
         where
```

And we then prove the induction principle stating that P holds for all of the natural numbers.

```
induction : ∀ n → P n
```

The details of the proof are not very illuminating but we include them for completeness' sake. We start by checking whether the natural number is zero, in which case we can use the base case, or whether it is a successor in which case we use the step case together with a recursive call to induction.

The stage has been set just right so that things compute where they should, impossible branches are self-evidently impossible and therefore the proof goes through. The thing to notice if we want to understand the proof is that the expression in the IsTrue instance argument gets smaller as we make more and more observations that constrain what the input natural number may be like.

```
induction [ ccs@(c :: cs) ] = checkZ (c =? 'Z') cs refl

  where

  checkS : ∀ {b} → Reflects c 'S' b → ∀ cs →
    {{@0 _ : IsTrue (b && ℕ cs)}} →
    ∀ {ccs} → c :: cs ≡ ccs .value → P ccs
  checkS true cs refl = PS [ cs ] (induction [ cs ])

  checkZ : ∀ {b} → Reflects c 'Z' b → ∀ cs →
    {{@0 _ : IsTrue (b && isNil cs || isS (c :: cs))}} →
    ∀ {ccs} → c :: cs ≡ ccs .value → P ccs
  checkZ true [] refl = P0
  checkZ false cs eq = checkS (c =? 'S') cs eq
```

An induction operator is of course not just one that has the right type but one that has the right computational behaviour too. We can readily check that our induction function behaves exactly like the primitive recursor on natural numbers ought to by writing two unit tests.

First, when applied to zero, the recursor immediately returns its base case.

```
_ : ∀ {P P0 PS} → induction P P0 PS zero ≡ P0
_ = refl
```

Second, when applied to the successor of a natural number n, the recursor returns its step case applied to n and the result of the recursive call.

```
_ : ∀ {P P0 PS n} →
    induction P P0 PS (suc n)
    ≡ PS n (induction P P0 PS n)
_ = refl
```

The fact that both of these unit tests are provable by refl means that Agda can tell by computation alone that the expressions are equal.

4.3 Example: Addition, Multiplication

As is well known, primitive recursion is enough to implement addition and multiplication on the natural numbers. So induction will be plenty enough power for us.

Addition of m to n can be implemented by induction on m. The base case, corresponding to zero + n, amounts to returning n. The step case amounts to taking the successor of the inductive hypothesis. This gives us the following definition:

```
_+_ : Elt ℕ → Elt ℕ → Elt ℕ
m + n = induction (λ _ → Elt ℕ) n (λ _ → suc) m
```

We can test the function thus implemented by writing a unit test reusing the constants defined in Section 3, checking for instance that 3 + 1 evaluates to 4.

```
_ : three + one ≡ four
_ = refl
```

Multiplication is defined in the same way: zero * n is equal to zero and the step case amounts to stating that (suc m) * n should evaluate to n + m * n.

```
_*_ : Elt ℕ → Elt ℕ → Elt ℕ
m * n = induction (λ _ → Elt ℕ) zero (λ _ → n +_) m
```

We can check with a unit test that our implementation verifies that $2 * 3$ equals $4 + 2$.

```
_ : two * three ≡ four + two
_ = refl
```

Because our induction function has the right computational behaviour, the definitions we just introduced should be well behaved too. They did pass a couple of unit tests but given that we are using a dependently typed host language we ought to do better than that.

5 Stringly Typed Proving

This section is dedicated to proving some of the properties of the functions we have defined. We hope to convince the reader that they could pick up any proof from the standard library's `Data.Nat.Properties` module and reproduce it on our stringly typed natural numbers.

5.1 Equality combinators

Now that we are entering serious proof territory, we will need to introduce some basic combinators witnessing the fundamental properties of propositional equality.

We use a block of variables Agda is authorised to implicitly quantify over to avoid repeating ourselves in this section.

```
variable
  A B : Set
  x y z : A
```

Propositional equality is a congruence. That is to say that if two values are equal, applying the same function to both will yield equal results.

```
cong : (f : A → B) → x ≡ y → f x ≡ f y
cong f refl = refl
```

We already know that propositional equality is a reflexive relation as witnessed by its constructor refl and we can additionally prove that is is a symmetric and transitive one.

```
sym : x ≡ y → y ≡ x
sym refl = refl

trans : x ≡ y → y ≡ z → x ≡ z
trans refl eq = eq
```

We now have the basic building blocks needed to build equality proofs.

5.2 Properties of Addition

Given our earlier observation that induction immediately returns its base case when applied to the natural number zero, it should not be any surprise that zero is trivially a left neutral for our definition of addition.

```
zero-+ : ∀ m → zero + m ≡ m
zero-+ m = refl
```

The proof that it is also a right neutral for addition requires a bit more work. We can use induction itself to build such a proof. The base case corresponding to zero + zero ≡ zero is trivially true. The step case is just a matter of using the induction hypothesis together with the fact that equality is a congruence to add a suc to both sides.

```
+-zero : ∀ m → m + zero ≡ m
+-zero =
  induction
    (λ m → m + zero ≡ m)
    refl
    (λ n → cong suc)
```

Similarly, our previous unit test should lead us to anticipate that the proof that the addition of suc *m* to *n* is equal to the successor of the addition of *m* to *n* is trival. This indeed holds true by computation alone.

```
suc-+ : ∀ m n → suc m + n ≡ suc (m + n)
suc-+ m n = refl
```

The statement stating that the addition of *m* to suc *n* is equal to the successor of the addition of *m* to *n* is however a bit trickier to deal with. It can once again be proven by using induction on *m*.

```
+-suc : ∀ m n → m + suc n ≡ suc (m + n)
+-suc m n =
  induction
    (λ m → (m + suc n) ≡ suc (m + n))
    refl
    (λ n → cong suc)
    m
```

These auxiliary lemmas are the intermediate results we need to be able to prove that addition is commutative. We, once again, proceed by induction and this time make crucial use of the fact that equality is symmetric and transitive.

```
+-comm : ∀ m n → m + n ≡ n + m
+-comm m n =
  induction
    (λ m → m + n ≡ n + m)
    (sym (+-zero n))
    (λ m ih → trans (cong suc ih) (sym (+-suc n m)))
    m
```

Let us conclude with one last example of a property one can prove of addition on stringly natural numbers: addition is associative.

```
+-assoc : ∀ m n p → (m + n) + p ≡ m + (n + p)
+-assoc m n p =
  induction
    (λ m → ((m + n) + p) ≡ (m + (n + p)))
    refl
    (λ m → cong suc)
    m
```

We have seen how we can define a type together with its induction principle, and how we can make use of this induction principle to program and prove our programs' properties. The next step is to use induction on a given type to define new types.

6 Our First Indexed Type: Fin

Given that the only type we have defined thus far is ℕ, we are going to use as the index of our type family. The natural candidate is Fin *n*, the type of finite numbers strictly smaller than *n*.

This definition proceeds by induction on the index and as such is characterised by a base and a step case.

```
Fin : Elt ℕ → Type
Fin = induction (λ _ → Type) base (λ _ → step)

  where
```

In the base case, corresponding to Fin zero, the boolean function is constantly false. The type is empty as there are no finite numbers strictly smaller than zero.

```
base : Type
base _ = false
```

In the step case, corresponding to Fin (suc *n*) we recognise a pattern similar to that used in the definition of ℕ: the string of interest is either 'Z'-headed with an empty tail or 'S'-headed with a tail of type Fin *n* (this type is provided to us by the induction hypothesis called *ih* here).

This time we do not bother introducing separate types for each of the constructors but we could very well do so.

```
step : Type → Type
step ih [] = false
step ih (c :: cs) = c == 'Z' && isNil cs
                 ||   c == 'S' && ih cs
```

We can once more define the basic constructors for this type. They have slightly more complex types, statically enforcing that the return index is suc-headed. "Z" gives rise to fzero.

```
fzero : ∀ {n} → Elt (Fin (suc n))
fzero {n} = Fin (suc n) ∋ "Z"
```

And extending an existing list of characters with 'S' is enough to compute the successor of a Fin *n* element as witnessed by fsuc.

```
fsuc : ∀ {n} → Elt (Fin n) → Elt (Fin (suc n))
fsuc [ k ] = [ 'S' :: k ]
```

The definition of the induction principle for Fin is left as an exercise to the reader. It is very similar to the definition of induction for ℕ. We will focus instead on a more interesting observation related to Fin.

6.1 Subtyping: Fin n <: ℕ

The astute reader will have noticed that the definition of Fin is not only similar to that of ℕ, it should be the case that all of the values of type Fin *n* are also stringly natural number.

This can actually be proven. It should be unsurprising by now that our tool of choice in this endeavour will be the induction principle for ℕ.

The key ingredient is the step case stating that, provided that we can already prove that elements of Fin *n* are elements of ℕ then we should be able to do the same for elements of Fin (suc *n*).

```
step : ∀ n → (∀ str → str ∈ Fin n → str ∈ ℕ) →
             (∀ str → str ∈ Fin (suc n) → str ∈ ℕ)
step n ih (c :: cs) isFin =
  checkZ (c =? 'Z') cs {{isFin}} where
```

We include the proof for completeness' sake even though it may not be illuminating for the Agda novice. It proceeds by case analysis on the input string, concluding immediately if it is "Z" and utilising the induction hypothesis if it is 'S'-headed instead.

```
checkS : ∀ {b} → Reflects c 'S' b → ∀ cs →
         {{IsTrue (b && Fin n cs)}} →
         (c :: cs) ∈ ℕ
checkS true cs {{isFin}} = ih cs isFin

checkZ : ∀ {b} → Reflects c 'Z' b → ∀ cs →
     {{IsTrue (b && isNil cs || c == 'S' && Fin n cs)}} →
     (c :: cs) ∈ ℕ
checkZ true [] = _
checkZ false cs = checkS (c =? 'S') cs
```

This can be put together with a trivial base case (remember that Fin zero is the empty type so it cannot have any element in it) to obtain the proof sub.

```
sub : ∀ n str → str ∈ Fin n → str ∈ ℕ
sub = induction
        (λ n → ∀ str → str ∈ Fin n → str ∈ ℕ)
        (λ _ ())
        step
```

This result allows us to write a cast function converting an element of Fin *n* into a stringly natural number. Notice that the value part is the identity. Given that the check part of the record will be erased at compile time this means we have defined a *zero cost coercion* from Fin *n* to ℕ which is much better than most state of the art dependently typed programming languages, save for Cedille [Diehl and Stump(2018)].

```
cast : ∀ {n} → Elt (Fin n) → Elt ℕ
cast {n} p .value = p .value
cast {n} p .check = sub n (p .value) (p .check)
```

7 Conclusion & Future Work

We have seen that we can take advantage of a dependently typed host language to seriously consider the prospect of safe and proven correct stringly typed programming.

We were able to define a notion of type of natural numbers carving out a subset of well structured strings. This type is closed under the usual constructors for the natural numbers zero and suc.

We then proved an induction principle for those strings that represent natural numbers. This empowered us to start programming over these stringly typed natural numbers in a way that is guaranteed total.

We demonstrated that our induction principle is strong enough to not only program on the stringly typed natural numbers but also to prove the fundamental properties of these programs.

We finally showed how we can use induction to define new types, and how we can take advantage of the fact we are doing dependent stringly typed programming to obtain zero cost coercions.

The definition of parametrised types such as the type of linked lists or binary trees with values stored at the leaves is left to future work.

References

[Diehl and Stump(2018)] L. Diehl and A. Stump. Zero-cost coercions for program and proof reuse. *CoRR*, abs/1802.00787, 2018. URL `http://arxiv.org/abs/1802.00787`.

[Dybjer(1994)] P. Dybjer. Inductive families. *Formal aspects of computing*, 6(4):440–465, 1994.

[Mahboubi and Tassi(2021)] A. Mahboubi and E. Tassi. *Mathematical Components*. Zenodo, Jan. 2021. doi: 10.5281/zenodo.4457887. URL `https://doi.org/10.5281/zenodo.4457887`.

[Norell(2009)] U. Norell. Dependently typed programming in Agda. In *AFP Summer School*, pages 230–266. 2009.