Towards Being Positively Negative about Dependent Types

Jan de Muijnck-Hughes

University of Strathclyde, UK
Jan.de-Muijnck-Hughes@strath.ac.uk

Negativity is Important for Proving and Programming Dependently typed programming languages, such as Idris2 [2] and Agda [3], provide expressive environments in which we can reason about and *run* our programs. Consider, for example, the following Idris2 definition of natural numbers (`Nat`) and two predicates (`NonZero` & `IsZero`) that state if a given natural number is non-zero or not:

```
data Nat = Z | S Nat     data NonZero : Nat -> Type     data IsZero : Nat -> Type
                           where NZ : NonZero (S n)       where IZ : IsZero Z
```

Decision procedures support reasoning about the correctness of `NonZero` and `IsZero`, by evidencing the construction of valid instances of the predicates or supplying a proof of falsity if not. These proofs are then repurposed as verified functions for programming. For example, consider the following 'proof' (`nonZero`) that `NonZero` is correct:

```
nonZero : (n : Nat) -> Dec (NonZero n)
nonZero Z = No absurd
  where
    absurd : NonZero Z -> Void
    absurd NZ impossible
nonZero (S x) = Yes NZ
```

```
data Dec : a -> Type where
  Yes : (prf : a) -> Dec a
  No  : (contra : a -> Void) -> Dec a
```

Our proof (`NonZero`) will produce an inhabitant of `NonZero` if the input *is* non-zero (`NZ`), or a proof of void if not (`absurd`). The generic datatype `Dec` encapsulates the result of `NonZero`, where `Yes` represents the positive result and `No` the negative result. If we wish, however, to use the negative result of `nonZero` at runtime then we cannot do so and determine why the function failed. Although it is self-evident that a negative result for `nonZero` implies that the input was non-zero, more complex predicates can fail for a variety of reasons.

Runtimes are Positive; Negativity is not Generally speaking, predicates are *positive* pieces of information and construction of decision procedures using `Dec` only enables runtime evidence to be produced when the 'happy' (positive) path is taken. All traces of *negative* information (i.e. falsity) are now a (careless) compile-time whisper. If we are *theorem proving* in dependently-typed languages, runtime reports of failure are not important; our code is proven correct. If we are *programming* in a dependently-typed language then knowing why a decision procedure failed is important; errors need to be reported. If we start to think more positively about being negative, we can start to report negative information more positively.

Being Constructive about Negation is a Positive The MSFP 2022 talk 'Data Types with Negation' discussed the idea of using *Constructive Negation* to rethink how we can work positively with negative information when programming [1]. Within dependently typed languages we can exploit constructive negation to rethink how we represent decidable decisions. For instance, consider the following *positive* definitions:

```
record Decidable where
  constructor D
  Positive : Type
  Negative : Type
  0 Cancelled : Positive -> Negative -> Void
```

```
0
Dec : Decidable -> Type
Dec (D positive negative no)
  = Either negative positive
```

`Decidable` is a datatype encapsulating the runtime irrelevant proof (identified by the `0` quantifier) that two positive pieces of information cancel each other out. Instances of `Decidable` are the input to a positive `Dec` which is translated to an instance of `Either`, where the positive truth is made `Right` and the positive evidence of falsity is `Left`. With `Decidable` and `Dec`, are decision procedures are now positive. We can illustrate our positive decisions by pairing our definitions of `IsZero` and `NonZero` together to produce `ISZERO`:

```
ISZERO : Nat -> Decidable
ISZERO n = D (IsZero n) (IsSucc n) prf
  where prf : IsZero n -> NonZero n -> Void
        prf IZ NZ impossible
```

```
isZero : (n : Nat) -> Dec (ISZERO n)
isZero Z     = Right IZ
isZero (S k) = Left  NZ
```

The function `prf` evidences that a number cannot be simultaneously zero and non-zero: our proof of falsity. The function `ISZERO` constructs the decidable predicate for a given natural number, and `isZero` is proof that we can decide if a number is non-zero or not. We can even reuse the same predicates, albeit flipped, to produce a decision procedure for `NONZERO`:

```
NONZERO : Nat -> Decidable
NONZERO n = D (IsSucc n) (IsZero n) prf
  where prf : NonZero n -> IsZero n -> Void
        prf NZ IZ impossible
```

```
nonZero : (n : Nat) -> Dec (NONZERO n)
nonZero Z     = Left  IZ
nonZero (S k) = Right NZ
```

Being Positive is Hard Through careful selection of our datatypes, we can develop the foundations of a 'positive' library for decision procedures. If we are not careful with our constructions we can easily end up with incorrect decisions being made. Take for example, quantifying over lists using 'All' and 'Any' predicates. The opposite of the All quantifier is the Any quantifier. Either all elements satisfy the positive predicate, or we will traverse the list until a positive instance of the negative predicate is found. The opposite of `Any` is, unfortunately, not a simple swapping of predicates as we saw with `NONZERO` and `ISZERO`. For the `Any` quantifier, we need to reverse the polarity of the predicates as well. A positive `Any` requires us to traverse the list and build the negative predicate until we find our positive one. The opposite of `Any` is that all items in the list produced negative predicates.

This Talk is about Positively Negative Programming In this talk, I will report *work-in-progress* that explores what it means to be *positively negative* when programming with dependent types. I will report how constructive negation re-frames not only the construction of a library[1] of positive decision procedures, but my experience using these procedures when programming. Specifically, I will report on reasoning about standard datatypes (natural numbers, lists, pairs, and strings) including their decidable equality, as well as their use when elaborating concrete syntax to intrinsically-typed terms for the Simply-Typed Lambda Calculus. Finally, I will examine how being so positive in one's negativity can reshape existing approaches to dependently-typed programming and report our decision procedures negativity more positively.

[1] https://github.com/jfdm/positively-negative

References

[1] Robert Atkey. 'Data Types with Negation'. Extended Abstract (Talk Only) at Ninth Workshop on Mathematically Structured Functional Programming, Munich, Germany, 2nd April 2022. 2022. URL: https://youtu.be/mZZjOKWCF4A.

[2] Edwin C. Brady. 'Idris 2: Quantitative Type Theory in Practice'. In: *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference)*. 2021, 9:1–9:26. DOI: 10.4230/LIPIcs.ECOOP.2021.9.

[3] The Agda Development Team. *Agda*. 2023. URL: https://github.com/agda/agda.