View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Repositorio Institucional Universidad EAFIT

Formalization of Programs with Positive Inductive Types

E. Lobo-Vesga

***Abstract*—Proof assistants are computer systems that allows a user to do mathematics on a computer helping with the development of formal proof by human-machine collaboration, however most of them only work with strictly positive types, this restriction limits the number of problem that can be formalized. This is perhaps the reason why verification of programs that use positive (and negative) types is uncommon. Hence, we use the programming logic created by Bove, Dybjer and Sicard-Ramírez that accept positive types to formalize the termination of a breadth-first search in a binary tree using continuations data type which is positive.**

***Keywords*—Inductive Types, Positive Types, Programming Logic, Continuations.**

I. Introduction

Types are ranges of significance of propositional functions [6] i.e. they are domains of predicates. For practice reasons we understand a type as a classification of data, and operations on them, which is useful to tell the compiler or interpreter how the programmer intends to use an specific data. Types supported by most programming languages include Booleans, Integers, Floating points numbers, Characters and Strings.

Each programming language has a form to represent and build types, then we say that a system or language has **inductive types** if we can create elements of a type with constants and functions of itself, for example, natural numbers using Peano's encoding can be represented as

```
data ℕ : Set where
  zero : ℕ
  suc  : (n : ℕ) → ℕ
```

where a natural number is created either from the constant "zero" or by applying the function "suc" to another natural number.

Inductive types can be represented as least fixed-points of appropriated functions (functors) [8]. For instance let 1 be the unity type and $+$ operator for the disjoint union, then the functor that represents the natural numbers is

$$\mathbb{N} = \mu X.1 + X$$

That is, if we have a type

```
data D : Set where
  lam : (D → D) → D
```

E. Lobo-Vesga is student of Mathematics Engineering, EAFIT University, Medellín, e-mail: elobove@eafit.edu.co.

its respective functor will be $D = \mu X.X \to X$. Based on that representations of inductive types as least fixed-points of a functor we can define **negative**, **positive** and **strictly positive** inductive types as follow: "The occurrence of a type variable is *positive* iff it occurs within an even number of left hand sides of $\to$-types, it is *strictly positive* iff it never occurs on the left hand side of a $\to$-type" [1]. In this context, the occurrence of a type variable is *negative* iff it occurs within an odd number of left hand sides of $\to$-types.

At this point we have a set of inductive types that can be classified as Negative or Positive and the positives can be Strictly positive or just Positives (see Fig. 1).

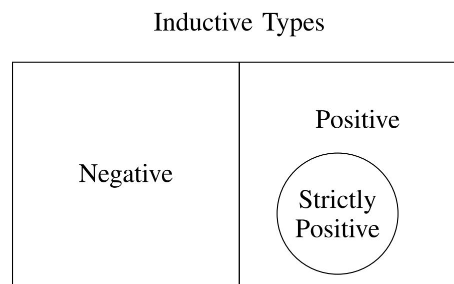

Fig. 1: Diagram sets of Inductive Types

Now, proof assistants are computer systems that allow a user to do mathematics on computer, helping with the development of formal proof by human-machine collaboration [2]. However most of them as Coq, Agda and Isabell only work with (or require) strictly positive inductive types. They do not use or accept negative types in order to avoid non-terminating functions, in other words, looping computation, and the positive (non-strictly positives) are exclude because they cannot be understood predicatively in general [1]. This constraint limits the number of programs that can be formalized.

Recently a programming logic where positive inductive types (as well as strictly positive ones) was developed by Bove, Dybjer and Sicard-Ramírez [7]. They built a computer-assisted framework, called Apia, for prove first-order theorems written in Agda using automatic theorem provers for first-order logic (ATPs).

We propose to identify and formalize some problem that make use of positive types using Apia.

II. Continuations as example

Basic concepts of continuations were discovered several times by different computers scientists in different contexts, for this reason, continuations were found useful for a variety of settings: “They underline a method of program transformations (into continuation passing-style), a style of definitional interpreter (defining one language by an interpreter written in another language), and a style of denotational semantics (in the sense of Scott ans Strachey)” [5]. For each setting, continuations respresent “the rest of the program” as a function or procedure.

We will understand continuations in the sense of Haskell’s continuations or Continuation Passing Style (CPS) which is a style of programming in which functions do not return values; rather, they pass control onto a *continuation*, which specifies what happens next. They are used to manipulate and alter the control flow of a program [3].

In 2000 Matthes uses continuations to do a breadth-first binary tree search [4]. In his example Matthes cites Hofmann’s unpublished work (Approaches to recursive data types - a case study, 1995) that defines the type of continuations as:

```
cont = D | C of (cont → list) → list
```

which is a non-strictly positive type because it occurs in the left hand of a →-type, but it is an interesting non-strictly positive type because it is a positive one.

III. Continuations in Agda

Matthes implements his example of continuations in the functional language SML, then we translated it to Haskell (see Appendix A) for understand the implementation.

Also, Matthes states several questions about the code and one of them is: “Does the program terminate for every input tree?”. We pretend to answer this question using Apia, but initially we implement the example in Agda to clarify why this cannot be formalized using it.

First of all we create a data type that represent a binary tree of natural numbers and continuations.

```
data Btree : Set where
  L : (x : ℕ) → Btree
  ℕ : (x : ℕ) (l r : Btree) → Btree

data Cont : Set where
  D : Cont
  C : ((Cont → List ℕ) → List ℕ) → Cont
```

As we said before, `Cont` is a non-strictly positive types and we need to use the flag `-no-positivity-check`, to use this type.

Later we implement four functions, `apply` and `breadth` are used to search in the binary tree; `ex` takes a continuation and generates a List of naturals, this function is used by `breadthfirst` that takes a binary tree, traverses it using the `breadth` function which result is passed to `ex` and it extracts the route of the search in a List.

```
apply : Cont → (Cont → List ℕ) → List ℕ
apply D     g = g D
apply (C f) g = f g

breadth : Btree → Cont → Cont
breadth (L x)     k =
  C $ λ g →x :: (apply k g)
breadth (ℕ x s t) k =
  C $ λ g →
  x :: (apply k (g ∘ breadth s ∘ breadth t))

ex : Cont → List ℕ
ex D     = []
ex (C f) = f ex

breadthfirst : Btree → List ℕ
breadthfirst t = ex (breadth t D)
```

Note that `ex` function is not structural recursive then we need to use `NO_TERMINATION_CHECK` pragma. Finally we created a binary natural as shown in Fig 2 and verify that the results of “breadthfirst” with this tree is the following list of naturals.

```
exList = [1,2,4,7,3,6,8,5,4,2,9]
```

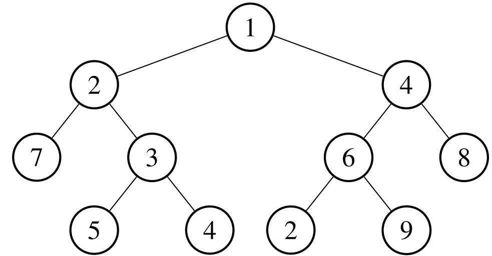

Fig. 2: Binary tree of natural numbers

All of the previous implementations and procedures were made in Agda and it type checked, we might think that this is enough to formalized that the program terminate, nevertheless we cannot conclude that because in our implementation we have to disable Agda’s termination checker with the flag `-no-positivity-check` and the pragma `NO_TERMINATION_CHECK`, this implies that our program is unsound when viewed as logic and also it weakens the reasoning that can be done about it [9].

IV. Continuations in Apia

Because definitions of `Cont` data type and `ex` function run afoul of Agda’s termination checker we intend to use Apia to implemented them and call ATPs as Vampire and E to prove properties of them.

Here we postulate a domain of terms and the term constructors using *higher-order abstract syntax* to represent the variable binding operator λ as Agda higher-order function.

```
postulate
  D          : Set
  zero [] d : D
  succ       : D → D
```

```
_○_ _::_    : D → D → D
lam         : (D → D) → D
node cont : D → D → D → D
```

Then we represent the inductive predicates N, ListN, Btree and Cont for total and finite natural numbers, list of natural numbers, binary tree of natural numbers and continuations respectively.

```
-- Natural numbers
data ℕ : D → Set where
  nzero : ℕ zero
  nsucc : ∀ {n} →ℕ n → ℕ (succ n)

-- List of Natural numbers
data Listℕ : D → Set where
  lnnil  : Listℕ []
  lncons : ∀ {n ns} →ℕ n → Listℕ ns →
    Listℕ (n :: ns)

-- Binary Nat Tree
data Btree : D → Set where
  Leaf : ∀ {x} →ℕ x → Btree x
  Node : ∀ {x l r} →ℕ x → Btree l →
    Btree r → Btree (node x l r)

-- Continuations
data Cont : D → Set where
  D' : Cont d
  C' : ∀ {x xs ys} →((Cont x →
    Listℕ xs) → Listℕ ys) →
    Cont (cont x xs ys)
```

Then with further work we may be able to implement `apply`, `breadth`, `ex` and `breadthfirst` functions and finally formalize that `breadthfirst` is (or not) a terminating functions.

V. Conclusion

The main goal of this research has been to identify and formalize a problem that make use of positive types (non-strictly positive) using the programming logic of Bove, Dybjer and Sicard-Ramírez. To achieve this goal, we have worked on different subjects that we present as main ideas of our work.

- Negative types could generate looping computations and Positive types cannot be understood predicatively in general.
- In AGDA when we use flags as `-no-positivity-check` or pragmas as `NO_TERMINATION_CHECK` we disable the AGDA's termination checker and the onus of create terminating functions is on the developer.
- Apia seems to be an useful framework to broaden the spectrum of programs that can be formalized.

Appendix A
Continuations in Haskell

```haskell
-- Binary tree
data Btree = L Int | N Int Btree Btree

-- Continuations : non-strictly positive
data Cont = D | C ((Cont -> [Int]) -> [Int])

apply :: Cont -> (Cont -> [Int]) -> [Int]
apply D     g = g D
apply (C f) g = f g

breadth :: Btree -> Cont -> Cont
breadth (L x)     k = C $ \g -> x : (apply k g)
breadth (N x s t) k = C $ \g -> x :
 (apply k (g . breadth s . breadth t))

-- Iteration on the data type Cont
ex :: Cont -> [Int]
ex D     = []
ex (C f) = f ex

breadthfirst :: Btree -> [Int]
breadthfirst t = ex $ breadth t D

-- Example
extree :: Btree
extree = N 1 (N 2 (L 7) (N 3 (L 5) (L 4)))
         (N 4 (N 6 (L 2) (L 9)) (L 8))

result :: [Int]
result = breadthfirst extree

exList :: [Int]
exList = [1,2,4,7,3,6,8,5,4,2,9]

ok :: Bool
ok = result == exList
```

References

[1] Abel, A. and Altenkirch, T. (2000). A Predicative Strong Normalisation Proof for a λ-Calculus with Interleaving Inductive Types, p. 21.

[2] Geuvers, H. (2009). Proof assistants: History, ideas and future. Sadhana Journal 34, pp 3-25.

[3] Haskell/Continations passing style. Retrieved from Wikibooks Web site: http://en.wikibooks.org/wiki/Haskell/Continuation_passing_style

[4] Matthes, R. (2000). Lambda Calculus: A Case for Inductive Definitions [PDF document]. Retrieved from Lecture Notes Online Web site: http://www.irit.fr/~Ralph.Matthes/papers/esslli.pdf

[5] Reynolds, J. C. (1993). The Discoveries of Continuations. LISP AND SYMBOLIC COMPUTATION: An International Journal 6, pp 233-247.

[6] Rusell, B. (1908). Mathematical logic as based on the theory of types. American Journal of Mathematics 30, pp 222-262.

[7] Sicard-Ramírez, A. (2014). Reasoning about Functional Programs by Combining Interactive and Automatic Proofs. Unpublished doctoral dissertion, University of the Republic, Uruguay.

[8] Sicard-Ramírez, A. (2014). Verification of Functional Programs [PDF document]. Retrieved from Lecture Notes Online Web site: http://www1.eafit.edu.co/asicard/courses/fpv-CB0683/slides/fpv-slides.pdf

[9] Weirich, S. and Casinghino, C. (2012). Generic Programming with Dependent Types. J. Gibbons (Ed.): Generic and Indexed Programming, LNCS 7470, pp 217–258.