Formalization of Programs with Positive Inductive Types

E. Lobo-Vesga

Abstract—Proof assistants are computer systems that allow 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

\[
\begin{align*}
\textbf{data } \mathbb{N} & : \text{Set where} \\
\text{zero} & : \mathbb{N} \\
\text{suc} & : (n : \mathbb{N}) \rightarrow \mathbb{N}
\end{align*}
\]

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

\[
\textbf{data } D : \text{Set where} \\
\text{lam} : (D \rightarrow D) \rightarrow D
\]

its respective functor will be \( D = \mu X . X \rightarrow X \). 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 \( \rightarrow \)-types, it is strictly positive iff it never occurs on the left hand side of a \( \rightarrow \)-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 \( \rightarrow \)-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).

![Diagram of Inductive Types](image)

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:

\[
\text{cont} = \text{D} \mid \text{C of (cont } \rightarrow \text{ list)} \rightarrow \text{ list}
\]

which is a non-strictly positive type because it occurs in the left hand of a \( \rightarrow \)-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.

\[
\text{data } \text{Btree} : \text{Set where}
\begin{align*}
L & : (x : \text{N}) \rightarrow \text{Btree} \\
N & : (x : \text{N}) \rightarrow (l \, r : \text{Btree}) \rightarrow \text{Btree}
\end{align*}
\]

\[
\text{data } \text{Cont} : \text{Set where}
\begin{align*}
D & : \text{Cont} \\
C & : ((\text{Cont } \rightarrow \text{ List N}) \rightarrow \text{ List N}) \rightarrow \text{ Cont}
\end{align*}
\]

As we said before, Cont is a non-strictly positive types and we need to use the flag \(-\text{no-positivity-check}\) 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.

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 \( \lambda \) as AGDA higher-order function.

\[
\text{postulate}
\begin{align*}
\text{D} & : \text{Set} \\
\text{zero} & : \text{D} \\
\text{succ} & : \text{D } \rightarrow \text{ D}
\end{align*}
\]
Then we represent the inductive predicates \( N, \text{ListN}, \text{Btree} \) and Cont for total and finite natural numbers, list of natural numbers, binary tree of natural numbers and continuations respectively.

**-- Natural numbers**

\[
\text{data } N : D \rightarrow \text{Set where}
\]

\[
\begin{align*}
nzero : & \ N \ 0 \\
n\text{succ} : & \ \forall \ {n} \rightarrow N \ n \rightarrow N \ (\text{succ} \ n)
\end{align*}
\]

**-- List of Natural numbers**

\[
\text{data } \text{ListN} : D \rightarrow \text{Set where}
\]

\[
\begin{align*}
\text{lnnil} : & \ \text{ListN} [] \\
\text{lncons} : & \ \forall \ {n \ ns} \rightarrow N \ n \rightarrow \text{ListN} \ ns \rightarrow \\
\text{ListN} : & \ (n :: ns)
\end{align*}
\]

**-- Binary Nat Tree**

\[
\text{data } \text{Btree} : D \rightarrow \text{Set where}
\]

\[
\begin{align*}
\text{Leaf} : & \ \forall \ {x} \rightarrow N \ x \rightarrow \text{Btree} \ x \\
\text{Node} : & \ \forall \ {x \ l \ r} \rightarrow N \ x \rightarrow \text{Btree} \ l \rightarrow \\
\text{Btree} \ r \rightarrow \text{Btree} \ (\text{node} \ x \ l \ r)
\end{align*}
\]

**-- Continuations**

\[
\text{data } \text{Cont} : D \rightarrow \text{Set where}
\]

\[
\begin{align*}
\text{D} : & \ \text{Cont} \ d \\
\text{C} : & \ \forall \ {x \ xs \ ys} \rightarrow ((\text{Cont} \ x \rightarrow \\
\text{ListN} \ xs \rightarrow \text{ListN} \ ys) \rightarrow \\
\text{Cont} \ (\text{cont} \ x \ xs \ ys))
\end{align*}
\]

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

**-- Binary tree**

\[
\text{data } \text{Btree} = L \ \text{Int} \ | \ N \ \text{Int} \ \text{Btree} \ \text{Btree}
\]

**-- Continuations : non-strictly positive**

\[
\text{data } \text{Cont} = D \ | \ C \ ((\text{Cont} \rightarrow [\text{Int}]) \rightarrow [\text{Int}])
\]

apply :: Cont \rightarrow (Cont \rightarrow [\text{Int}]) \rightarrow [\text{Int}]

apply D g = g D

apply (C f) g = f g

breadth :: Btree \rightarrow Cont \rightarrow Cont

breadth (L x) k = C $ \ g \rightarrow x : (apply \ k \ g)

breadth (N x s t) k = C $ \ g \rightarrow x : 

(apply \ k \ (g . \ \text{breadth \ s} . \ \text{breadth} \ t))

**-- Iteration on the data type Cont**

\[
\begin{align*}
ex : & \ \text{Cont} \rightarrow [\text{Int}] \\
ex D = & \ [] \\
(ex \ (C \ f) = f \ ex)
\end{align*}
\]

breadthfirst :: Btree \rightarrow [\text{Int}]

\[
\text{breadthfirst} \ t = \ ex \$ \ \text{breadth} \ t \ D
\]

**-- Example**

\[
\text{extree} :: \text{Btree}
\]

\[
\text{extree} = N \ 1 (N \ 2 (L \ 7) (N \ 3 (L \ 5) (L \ 4)))
\]

\[
(N \ 4 (N \ 6 (L \ 2) (L \ 9)) (L \ 8))
\]

\[
\text{result} :: [\text{Int}]
\]

\[
\text{result} = \text{breadthfirst} \ \text{extree}
\]

\[
\text{exList} :: [\text{Int}]
\]

\[
\text{exList} = [1,2,4,7,3,6,8,5,4,2,9]
\]

\[
\text{ok} :: \text{Bool}
\]

\[
\text{ok} = \text{result} == \text{exList}
\]

References


