Formalization of Programs with Positive Inductive Types

Elisabet Lobo-Vesga
Andrés Sicard-Ramírez

Universidad EAFIT
Medellín, Colombia

February 2015

Basic Concepts

- A **type** is a classification of data and operations on them [1]
- A system/language has **inductive types** if we can create elements of a type with constants and functions of itself

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

- Inductive types can be represented as least fixed-points of appropriated functions (functors)[1]

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

[1] Sicard-Ramírez, A. (2014). Verification of Functional Programs. http://www1.eafit.edu.co/asr/courses/fpv-CB0683/slides/fpv-slides.pdf

- If we have a type

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

with his functor $D = \mu X.X \to X$ we can classify D as a **negative**, **positive** or **strictly positive** type 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."[2]

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

Basic Concepts

- Positive

```
data A : Set where
  conA : A → X → X → A
```

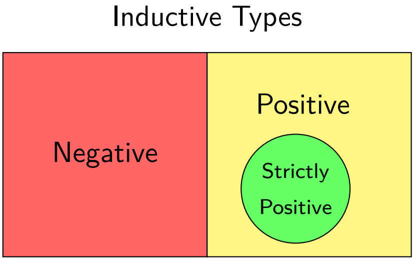
- Negative

```
data B : Set where
  conB : (B → B) → B
```

- Strictly Positive

```
data C : Set where
  conC : X → Y → C
```

- Proof assistants require strictly positive inductive types to avoid non-terminating functions
- Real world problems use non-strictly positive types, however verification of them is uncommon.



APIA to the Rescue

What do we propose?

To identify and formalize some problem that make use of positive inductive types using the programming logic of A. Bove, P. Dybjer and A. Sicard-Ramírez which support positive inductive types.[3]

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 Unpublished doctoral dissertion, University of the Republic, Uruguay.

Definition

Continuation Passing Style (CPS) 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.[4]

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

Breadth-first search

In 2000 Matthes uses continuations to do a breadth-first binary tree search[5]. 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:

```
data Cont = D | C ((Cont → [Int]) → [Int])
```

Q: Does the program terminate for every input tree?

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

Data types

```
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
```

We use the flag `-no-positivity-check` to work with non-strictly positive types.

Functions

```
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))
```

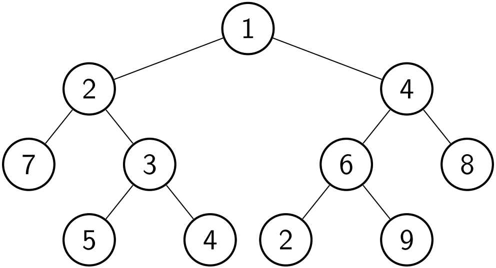
Functions

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

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

We use NO_TERMINATION_CHECK pragma to work with non structural recursive function.

Example




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

Problems

Although our implementation type-checked we cannot conclude that the program terminates because we use 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[6].

[6] Weirich, S. and Casinghino, C. (2012). Generic Programming with Dependent Types. pp 217–258.

Postulates

We postulate a domain of terms and the term constructors

```
postulate
  D            : Set
  zero [] d    : D
  succ         : D → D
  _○_ _∷_      : D → D → D
  lam          : (D → D) → D
  node cont    : D → D → D → D
```

Inference rules

We declare the unary predicates ℕ and `Listℕ` whith their introduction rules.

```
-- 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)
```

Inference rules

We declare the unary predicates Btree and Cont whith their introduction rules.

```
-- 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)
```

Problems

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

```
breadthfirst : ∀ {t} ∃ [ xs ] →
  Btree t →  Listℕ xs
```