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

```plaintext
data \( \mathbb{N} : \text{Set} \) where
  zero : \( \mathbb{N} \)
  suc   : (n : \mathbb{N}) \to \mathbb{N}
```

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

\[ \mathbb{N} = \mu X.1 + X \]

If we have a type

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

with his functor \( D = \mu X. X \rightarrow X \) 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 \( \rightarrow \)-types, it is strictly positive iff it never occurs on the left hand side of a \( \rightarrow \)-type.”^2

Basic Concepts

- **Positive**
  \[
  \text{data } A : \text{ Set where }
  \text{conA : } A \rightarrow X \rightarrow X \rightarrow A
  \]

- **Negative**
  \[
  \text{data } B : \text{ Set where }
  \text{conB : } (B \rightarrow B) \rightarrow B
  \]

- **Strictly Positive**
  \[
  \text{data } C : \text{ Set where }
  \text{conC : } X \rightarrow Y \rightarrow C
  \]
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.

**Inductive Types**

- **Negative**
- **Positive**
  - **Strictly Positive**
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.\(^3\)

---

Definition
Continuation Passing Style (CPS) is a style of programming in which functions do not return values; rather, they pass control onto a \textit{continuation}, which specifies what happens next. They are used to manipulate and alter the control flow of a program.\footnote{Haskell/Continuations 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:

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

Q: Does the program terminate for every input tree?

---

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 \texttt{-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 o breadth s o breadth t))
Functions

\[
\text{ex} : \text{Cont} \rightarrow \text{List } \mathbb{N} \\
\text{ex} \ D &= [] \\
\text{ex} \ (C \ f) &= f \ \text{ex}
\]

\[
\text{breadthfirst} : \text{Btree} \rightarrow \text{List } \mathbb{N} \\
\text{breadthfirst} \ t &= \text{ex} \ (\text{breadth} \ t \ D)
\]

We use \texttt{NO\_TERMINATION\_CHECK} pragma to work with non structural recursive function.
Example

\[
\text{exList} = [1, 2, 4, 7, 3, 6, 8, 5, 4, 2, 9]
\]
Problems
Although our implementation type-checked we cannot conclude that
the program terminates because we use the flag \texttt{-no-positivity-check}
and the pragma \texttt{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\textsuperscript{6}.

Postulates

We postulate a domain of terms and the term constructors

\[
\text{postulate}
\]

\[
\begin{align*}
D & : \text{Set} \\
\text{zero} [\ ] & : D \\
\text{succ} & : D \rightarrow D \\
_\circ_ & : D \rightarrow D \rightarrow D \\
\text{lam} & : (D \rightarrow D) \rightarrow D \\
\text{node cont} & : D \rightarrow D \rightarrow D \rightarrow D
\end{align*}
\]
Inference rules
We declare the unary predicates $\mathbb{N}$ and $\text{List}\mathbb{N}$ with their introduction rules.

\begin{itemize}
  \item \textit{Natural numbers}
  \begin{verbatim}
  data \text{N} : D \to \text{Set} where
    nzero : \text{N} zero
    nsucc : \forall \{n\} \to \text{N} n \to \text{N} (\text{succ} n)
  \end{verbatim}

  \item \textit{List of Natural numbers}
  \begin{verbatim}
  data \text{ListN} : D \to \text{Set} where
    lnnil : \text{ListN} []
    lncons : \forall \{n ns\} \to \text{N} n \to \text{ListN} ns \to \\
    \text{ListN} (n :: ns)
  \end{verbatim}
\end{itemize}
Inference rules
We declare the unary predicates Btree and Cont with 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.

\[
breadthfirst : \forall \{t\} \exists [\ xs ] \rightarrow
\text{Btree } t \rightarrow \text{List}\mathbb{N} \ xs
\]