# Generic programming for (co)inductive types in Type Theory

**Marcin Benke**

**Chalmers Technical University**

**Makoto Takeyama**

**AIST Amagasaki, Japan**

**Motivation:** reasoning about state transition systems — of interest to AIST industrial partners.

**Related work**

Gimenéz 1996 *A Calculus of Infinite Constructions and its application to the verification of communicating systems*.

Benke, Dybjer, Jansson 2003 *Universes for generic programs and proofs in dependent type theory*

# Generic functional programming

- **Basic idea:** define generic functions by induction on the definition of a data type
- **Examples:**
  - generic Boolean equality: SML (built-in) and Haskell (derivable class)
  - generic **map** combinators, and generic **iteration** and **recursion** over (co)inductive datatypes
- **Benefits**:
  - highly reusable and adaptive definitions
  - well suited for building libraries of programs, theorems **and proofs**

# Dependent types

**Examples**:

- Vect $n$ — vectors (lists) of length $n$
- data structures with invariants: ordered lists, balanced trees, AVL-trees, red-black-trees, etc.
- In general: we can express more or less arbitrary properties of programs and data structures.
- Universe of codes for datatypes — the natural setting for generic programming

The ideas presented in this talk have been implemented and tested using the *Alfa* proof editor.

# Universes

- A universe consists of
  - a set of codes for datatypes: Sig : Set
  - a decoding function: $\mathrm{T} : (\Sigma : \mathrm{Sig}) \to \mathrm{Set}$
- **Example**: a universe for single-sorted algebras is described by the set of signatures $\mathrm{Sig} = [\mathrm{Nat}]$, and the decoding function $T$ which maps a signature to (the carrier of) its term algebra.
- the booleans have signature $[0, 0]$, the natural numbers — $[0, 1]$,

# A universe for inductive families

The set of codes for inductive families indexed by $I$:

$$\begin{array}{lcl} \mathsf{Sig}_I & = & [\mathsf{Arity}_I] \\ \mathsf{Arity}_I & = & \textbf{data Nil}\ (i : I) \mid \textbf{NonRec}\ (A : \mathsf{Set})(A \to \mathsf{Sig}_I) \mid \textbf{Rec}\ I\ \mathsf{Sig}_I \end{array}$$

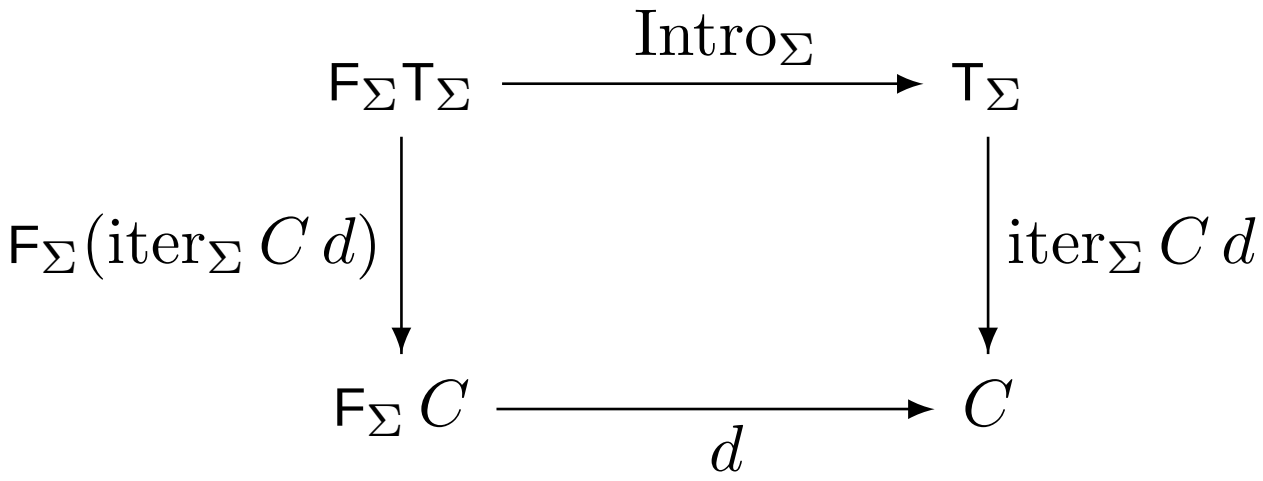
Example: vectors

$$\frac{n \in \mathsf{Nat} \quad A \in \mathsf{Set}}{\mathit{Vec}\ n\ A \in \mathsf{Set}} \qquad \frac{}{\mathit{Vnil} \in \mathit{Vec}\ 0\ A} \qquad \frac{x \in A \quad xs \in \mathit{Vec}\ n\ A}{\mathit{Vcons}\ x\ xs \in \mathit{Vec}\ (n+1)\ A}$$

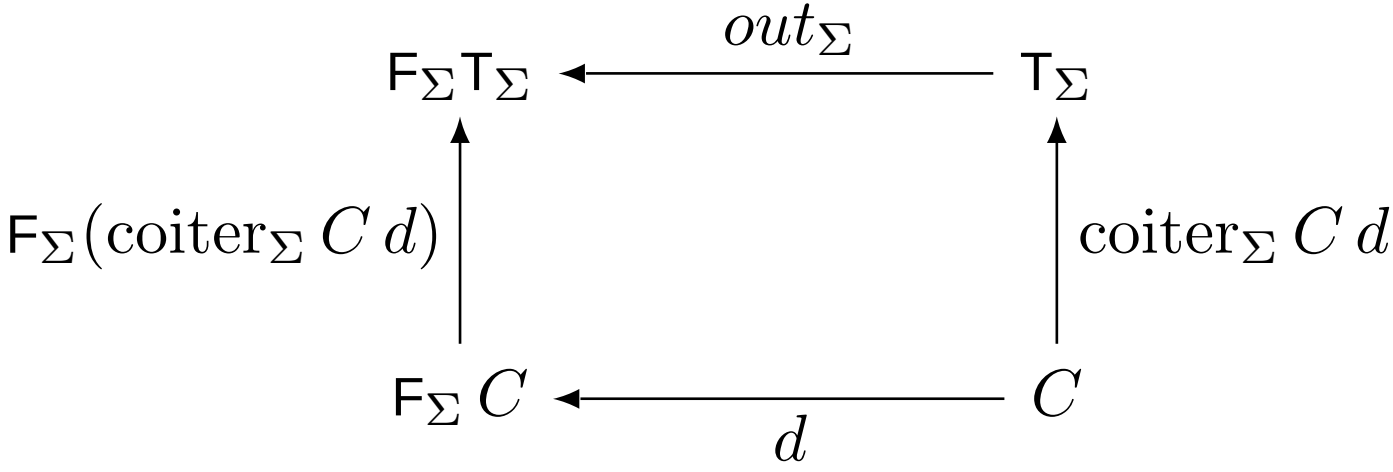
The signature for vectors:

$$[\textbf{Nil}\ 0, \quad \textbf{NonRec}\ A\ \lambda\_.\textbf{NonRec}\ \mathsf{Nat}\ \lambda n.\textbf{Rec}\ n\ \textbf{Nil}\ (S\ n))]$$

A generic iterator is defined basing on the initial algebra diagram:




For most universes, instead of initial algebras and iterators, one can consider final coalgebras and coiterators



# Universes for coinductive types

- The dual of inductive types — coinductive types, enable us to build and reason about infinite structures, e.g. streams.
- Gimenez in his *Calculus of Infinite Constructions* lays foundations for coinductive types in type theory and gives some proofs in Coq (including Park's coinduction principle for streams).
- For many universes, we can get coinductive types by slightly altering the decoding function.

# Some examples

By defining functions mapping transition systems to codes we can define e.g.

- the set of all possible runs (traces)
- the tree containing all possible runs
- a set of runs fulfilling certain safety criteria
- a proof that all runs satisfy safety criteria

By mapping codes to codes, we can, define generically inductive families for equality and bisimulation.

Having done that we generalize Gimenez’s results by proving generic Park’s principle

# Conclusions

- Better representation for inductive families
- Constructing universes for coinductive types is no more difficult in the inductive case. . .
- . . . but using them might be much more difficult
- Still there is hope for applying these techniques for reasoning about transition systems
- Work in progress. . .

# Thank you!

# Tree of all runs

```
State::Set
trans :: State -> [State]

star’ (s::State)(ss::[State]) :: Arity State
    = case ss of {
        [] -> Nil s;
        (x:xs) -> Rec x (star’ s xs);}
star (s::State) :: Arity State
star s = star’ (trans s)

stSig :: Sig State
stSig = [NonRec State star]
```

# Park's principle

```
package B = BaseConst
package Park(fi::B.Sig) where
  T2  = And (B.T fi) (B.T fi)
  SEqT (fi::B.Sig):: Sig T2 -- this one is hairy
  EqT = T T2 (SEqT fi)
  Rel::Type = T2 -> Set

  RHom(R,S::Rel)::Set = (i::T2)-> R i -> S i

  princ(R::Rel)::RHom R (FEq fi R) -> RHom R EqT
                  = CoIt T2 (SEqT fi) R
```