Induction and Co-induction
Well-founded recursion

- Well-foundedness implies termination of some sort
- No infinite loops
- Two dual flavors: induction and co-induction
**Induction**

\[
\begin{align*}
\text{data Nat where} & \quad \text{data List } a \text{ where} \\
& \hspace{1cm} Z : \text{Nat} \quad \quad \quad \quad \quad \quad \quad \text{Nil} : \text{List } a \\
& \hspace{1cm} S : \text{Nat} \to \text{Nat} \quad \quad \text{Cons} : a \to \text{List } a \to \text{List } a
\end{align*}
\]

\[
\begin{align*}
\text{length} & : \forall a. \text{List } a \to \text{Nat} \\
\text{length } \text{Nil} & = Z \\
\text{length } (\text{Cons } x \ xs) & = \text{let } y = \text{length } xs \text{ in } S y
\end{align*}
\]
Co-induction

```coq
codata InfList a where
  Cons : a → InfList a → InfList a

  zeroes : InfList Nat
  zeroes = Cons Z zeroes

  count : Nat → InfList Nat
  count x = Cons x (count S(x))
```
Co-induction

\textbf{codata} \texttt{Stream }a\texttt{ where}

\texttt{Head : Stream }a\rightarrow a

\texttt{Tail : Stream }a\rightarrow \texttt{Stream }a

\texttt{zeroes : Stream Nat}

\texttt{zeroes.}\texttt{Head }= \texttt{Z}

\texttt{zeroes.}\texttt{Tail }= \texttt{zeroes}

\texttt{count : Nat }\rightarrow \texttt{Stream Nat}

\texttt{count }x\texttt{.}\texttt{Head }= x

\texttt{count }x\texttt{.}\texttt{Tail }= \texttt{count }\texttt{(x }+ 1\texttt{)}
Well-founded induction and co-induction

- Well-foundedness for induction is clear
  - Structural induction

- Well-foundedness for co-induction is murky
  - Productivity? Guardedness?

- Asymmetric bias for induction over co-induction

- Can they be unified?

- Idea: Complete symmetry to find structure
Recursion on Structures
Classical sequent calculus: a symmetric language

- Producers (terms):
  \[ v \in Term ::= x \mid \mu \alpha . c \mid \ldots \]

- Consumers (co-terms):
  \[ e \in CoTerm ::= \alpha \mid \tilde{\mu} x . c \mid \ldots \]

- Computations (commands):
  \[ c \in Command ::= \langle v \parallel e \rangle \]
Input and output

A place for everything and everything in its place.

- Computations do not return, they run
- Unspecified inputs \((x, y, z)\) and outputs \((\alpha, \beta, \gamma)\)
- \(\tilde{\mu}\) abstracts over unspecified input
  \[\langle x \| \tilde{\mu} y . c \rangle = c\{y/x\}\]
- \(\mu\) abstracts over unspecified output
  \[\langle \mu \beta . c \| \alpha \rangle = c\{\beta/\alpha\}\]
Data types

- Values are *constructed*
- Consumed by *pattern matching*

\[
\begin{align*}
\textbf{data Nat} \ & \ 	extbf{where} \ \\
& Z : \vdash \text{Nat} | \\
& S : \text{Nat} \vdash \text{Nat} |
\end{align*}
\]

\[
\begin{align*}
\textbf{data List}(a) \ & \ 	extbf{where} \ \\
& \text{Nil} : \vdash \text{List}(a) | \\
& \text{Cons} : a, \text{List}(a) \vdash \text{List}(a) |
\end{align*}
\]
Co-data types

- Observations are constructed
- Produced by *pattern matching*

\[ \text{codata } a \rightarrow b \text{ where} \]
\[ _\cdot_ : \quad a \mid a \rightarrow b \vdash b \]

\[ \text{codata } \text{Stream}(a) \text{ where} \]
\[ \text{Head} : \quad \mid \text{Stream}(a) \vdash a \]
\[ \text{Tail} : \quad \mid \text{Stream}(a) \vdash \text{Stream}(a) \]
User-defined (co-)data types

- All types user-definable, follow same pattern
- ADTs from functional languages are data
- Functions are co-data
- Universal quantification is co-data
  - Explicit $\forall$ à la System $F_\omega$
- Existential quantification is data
- Types that lie outside the functional paradigm
Recursion on data structures

Called function

\[ \langle \text{length} \parallel xs \cdot \alpha \rangle \]

Have List\((a)\) \quad \text{Want Nat}

\[ \langle \text{length} \parallel \text{Nil} \cdot \alpha \rangle = \langle Z \parallel \alpha \rangle \]
\[ \langle \text{length} \parallel \text{Cons}(x, xs) \cdot \alpha \rangle = \langle \text{length} \parallel xs \cdot \tilde{\mu}y.\langle S(y) \parallel \alpha \rangle \rangle \]
Recursion on co-data structures

Called function

\[
\langle \text{count} \parallel x \cdot \alpha \rangle
\]

Have Nat \quad \text{Want Stream(Nat)}

\[
\langle \text{count} \parallel x \cdot \text{Head}[^{\alpha}] \rangle = \langle x \parallel \alpha \rangle
\]

\[
\langle \text{count} \parallel x \cdot \text{Tail}[^{\alpha}] \rangle = \langle \text{count} \parallel \text{S}(x) \cdot \alpha \rangle
\]
Structural recursion

- Distinction between induction and co-induction fade away

- Both are modes of recursion on *some* structure
  - Induction: recurse on data structure value
  - Co-induction: recurse on co-data structure observation

- Recursive invocations run with sub-structures

\[
\langle \text{length}\|\text{Cons}(x, xs)\cdot \alpha \rangle = \langle \text{length}\|xs \cdot \tilde{\mu}y.\langle S(y)\|\alpha \rangle \rangle
\]
\[
\langle \text{count}\|x \cdot \text{Tail}[\alpha] \rangle = \langle \text{count}\|S(x) \cdot \alpha \rangle
\]
Structures for Recursion
Finding the sub-structure

- To check well-foundedness, check for decreasing sub-structure

- But relevant sub-structure appears inside a larger structural context

\[
\langle \text{length} \parallel \text{Cons}(x, xs) \cdot \alpha \rangle = \langle \text{length} \parallel xs \cdot \tilde{\mu} y. \langle S(y) \parallel \alpha \rangle \rangle
\]
\[
\langle \text{count} \parallel x \cdot \text{Tail}[\alpha] \rangle = \langle \text{count} \parallel S(x) \cdot \alpha \rangle
\]

- Structure of function calls not special, same for tuples, etc.

- How do we know where to find it?
Tracking sub-structures with sized types

- Type-based approach to termination
- Size approximate the depth of structures
- Types can be indexed by (several) sizes
- Separate recursion in types from recursion in programs
Recursion in types

- Add extra size index to recursive (co-)data types
- Change in size tracks recursive sub-structures of recursive types
- Given \( x : \text{Nat}(i) \) then \( S(x) : \text{Nat}(i + 1) \)
- Given \( \alpha : \text{Stream}(i, a) \) then \( \text{Tail}[\alpha] : \text{Stream}(i + 1, a) \)
Recursion in programs

- Recursion over structures of recursive type quantifies over size index

- \( \text{length} : \forall a. \forall i. \text{List}(i, a) \rightarrow \text{Nat}(i) \)

- \( \text{count} : \forall i. (\exists j. \text{Nat}(j)) \rightarrow \text{Stream}(i, \exists j. \text{Nat}(j)) \)

- Different kinds of sizes for different purposes:
  - Step-by-step (primitive) recursion: computation depends on type-level size index at run-time, dependently typed vectors
  - Bounded (noetherian) recursion: type-level size index is erasable at run-time, recurse on deeply nested sub-structure
Structures for structural recursion

- Size quantifiers are themselves (co-)data types
- Their values and observations are structures for specifying structural recursion
- Like $\forall$ and $\exists$, quantify sizes over arbitrary types
- Can “induct” over co-data types, vice versa
  - Eliminate the need for strictures on structures
More in the paper

- Source effect-free functional calculus with recursion, data types, and “pure” objects
- Target classical calculus with user-defined recursive (co-)data and recursion schemes
- Modest dependent types with control effects
- Different evaluation strategies, parametrically
- Strong normalization
- Type erasure and computationally relevant types
Final thoughts

- Induction and co-induction are modes of structural recursion
- Find the structure with both sides of the story
- Duality and symmetry are powerful weapons: they invert murky problems into clear ones