Ramified Structural Recursion and Corecursion

Jim Royer

Syracuse University

12 November 2013

Joint work with Norman Danner (Wesleyan University)

Goals: General

1. Study "poly-time" computation over data and codata.*
   (Data? Codata?? Definitions shortly.)
2. Proceed synthetically via restricted programming formalisms.
3. Be as simple and as general as we can manage.
4. Solve some problems. / Find new problems. / Explore!

Non-goals (for *this* paper)

- ✘ Finding the "one true" notion of poly-time over data and codata.
- ✘ Delving too deep into higher-types.

* Turing '36 *and* Hartmanis and Stearns '65 concern computation over streams.

Goals: Technical

1. Keep the formalism **as standard & straightforward as possible**
   . . . and see how far these choices carry us.
2. Build a platform for further exploration.
3. Avoid *ad hoc* choices and inventions!
4. . . . unless
   - we are driven to make a choice (), or
   - we need to protect goal 1.

The foundation layer

The foundation layer

L: a simply typed lambda calculus

Syntax $E ::= () \mid (E, E) \mid (\pi_1\, E) \mid (\pi_2\, E)$
$\mid (\iota_1\, E) \mid (\iota_2\, E) \mid \textbf{case}\, E\, \textbf{of}\, (\iota_1\, X) \Rightarrow E;\ (\iota_2\, X) \Rightarrow E$
$\mid X \mid (\lambda X.E) \mid (E\ E)$

$X ::= \textit{identifiers}$

Types $T ::= \textbf{unit} \mid T \times T \mid T + T \mid T \to T$

- $()$ ≡ the 0-tuple, only inhabitant of **unit**, & **unit** only L-base-type
- $+$ ≡ tagged disjoint union, $\iota_1 : A_1 \to A_1 + A_2$, $\iota_2 : A_2 \to A_1 + A_2$
- The L-types have standard set-theoretic interpretations.

- simple type $=_{\text{def}}$ a type build from $+$, $\times$, $\to$, and base types

The foundation layer
L: a simply typed lambda calculus
Syntax $E ::= () \mid (E, E) \mid (\pi_1\, E) \mid (\pi_2\, E) \mid (\iota_1\, E) \mid (\iota_2\, E) \mid \textbf{case}\, E\, \textbf{of}\, (\iota_1\, X) \Rightarrow E;\ (\iota_2\, X) \Rightarrow E \mid X \mid (\lambda X.E) \mid (E\ E)$
$X ::= \textit{identifiers}$
Types $T ::= \textbf{unit} \mid T \times T \mid T + T \mid T \to T$
- $()$ ≡ the 0-tuple, only inhabitant of **unit**, & **unit** only L-base-type
- $+$ ≡ tagged disjoint union, $\iota_1 : A_1 \to A_1 + A_2$, $\iota_2 : A_2 \to A_1 + A_2$
- The L-types have standard set-theoretic interpretations.
- simple type $=_{\text{def}}$ a type build from $+$, $\times$, $\to$, and base types

- CCC's
- ground type = level 0 type
- Each L type has finitely many inhabitants.
- Next: supply L with something to compute over.

"Classical" formalism #1: $S^- = L +$ inductive data

Inductive data declarations: **data** $\tau = \mu t.\sigma$

Examples

- **data nat** $= \mu t.(\textbf{unit} + t)$

  Alt: **data nat** $=$ Zero **of unit** ‖ Succ **of nat**
  Elms: Zero, Succ(Zero), Succ(Succ(Zero)), ...

- **data tree** $= \mu t.(\textbf{unit} + t \times t)$

  Alt: **data tree** $=$ Leaf **of unit** ‖ Fork **of tree** $\times$ **tree**
  Elms: Leaf, Fork(Leaf, Leaf), Fork(Fork(Leaf, Leaf), Leaf), ...

- **data natLst** $= \mu t.(\textbf{unit} + \textbf{nat} \times t)$

  Alt: **data natLst** $=$ Null **of unit** ‖ Cons **of nat** $\times$ **natLst**
  Elms: Null, Cons(Zero, Null), Cons(Succ(Zero), Cons(Zero, Null)), ...

"Classical" formalism #1: $S^- = L +$ inductive data

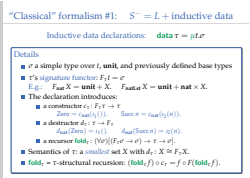
Inductive data declarations: **data** $\tau = \mu t.\sigma$

Details

- σ a simple type over t, **unit**, and previously defined base types
- τ's signature functor: $F_\tau t = \sigma$
  E.g.: $F_{\textbf{nat}}\, X = \textbf{unit} + X$. $F_{\textbf{natLst}}\, X = \textbf{unit} + \textbf{nat} \times X$.
- The declaration introduces:
  - a constructor $c_\tau : F_\tau \tau \to \tau$
    $\text{Zero} = c_{\textbf{nat}}(\iota_1())$. $\text{Succ}\, n = c_{\textbf{nat}}(\iota_2(n))$.
  - a destructor $d_\tau : \tau \to F_\tau$
    $d_{\textbf{nat}}(\text{Zero}) = \iota_1()$. $d_{\textbf{nat}}(\text{Succ}\, n) = \iota_2(n)$.
  - a recursor **fold**$_\tau$: $(\forall\sigma)[(F_\tau\sigma \to \sigma) \to \tau \to \sigma]$.
- Semantics of τ: a *smallest* set X with $d_\tau : X \cong F_\tau X$.
- **fold**$_\tau$ = τ-structural recursion: $(\textbf{fold}_\tau f) \circ c_\tau = f \circ F(\textbf{fold}_\tau f)$.

Ramified Structural Recursion and Corecursion

└"Classical" formalism #1: $S^- = L +$ inductive data



Other notions of data include

- mutually recursive types
- parameterized defs
- and more type constructors E.g.,
  - $X \mapsto X^*$ — finite lists
  - $X \mapsto \mathcal{P}_{\text{fin}}(X)$ — finite power sets
  - $X \mapsto (W \to X)$ — functions from prior types
- See Adámek, Milius, & Moss and Rutten for more examples

"Classical" formalism #1: $S^- = L +$ inductive data

Recall: $\textbf{data}\ \textbf{nat} = \mu t.(\textbf{unit} + t)$. $\quad \mathsf{Zero} = c_{\textbf{nat}}(\iota_1())$. $\quad \mathsf{Succ}\, x = c_{\textbf{nat}}(\iota_2(x))$.

Example: $plus : \textbf{nat} \to \textbf{nat} \to \textbf{nat}$

$$plus\, x\, y = \textbf{fold}_{\textbf{nat}}\ \Big(\lambda z.\ \textbf{case}\ z\ \textbf{of}\ (\iota_1\, w) \Rightarrow y;\ (\iota_2\, w) \Rightarrow (c_{\textbf{nat}}(\iota_2\, w))\Big)\ x$$
$$= \textbf{let}\ f(\iota_1\, w) = y; f(\iota_2\, w) = (\mathsf{Succ}\, w)\ \textbf{in}\ (\textbf{fold}_{\textbf{nat}}\, f\, x)$$

Using $(\textbf{fold}_\tau f) \circ c_\tau = f \circ F(\textbf{fold}_\tau f)$, the definition of f above, etc.:

$$\boxed{plus\ \mathsf{Zero}\ y} = \textbf{fold}_{\textbf{nat}}\ f\ \mathsf{Zero} \qquad = \boxed{y}$$

$$\boxed{plus\ (\mathsf{Succ}\ x)\ y} = \textbf{fold}_{\textbf{nat}}\ f\ (\mathsf{Succ}\ x) = \mathsf{Succ}(\textbf{fold}_{\textbf{nat}}\ f\ x) = \boxed{\mathsf{Succ}(plus\, x\, y)}$$

$\textbf{fold}_\tau$ = structural/primitive recursion on τ-data

"Classical" formalism #1: $S^- = L +$ inductive data

More examples (sugared and not)

// times $x\ y = x \cdot y$

$times\, x\ \mathsf{Zero} = \mathsf{Zero} \qquad times\, x\, (\mathsf{Succ}\, y) = plus\, x\, (times\, x\, y)$

// sqrLst $[n_0, \ldots, n_k] = [n_0^2, \ldots, n_k^2]$

$sqrLst\ \mathsf{Null} = \mathsf{Null} \qquad sqrLst\ (\mathsf{Cons}(x, ys)) = \mathsf{Cons}((times\, x\, x), sqrLst\ ys)$

// The Péter-Robinson-Ackermann function (with very little sugar)

$acker\, m\, n$

$= \textbf{let}*\ iter\, f\, k = \textbf{let}\ g(\iota_1()) = (f\ (\mathsf{Succ}\ \mathsf{Zero}));\ g(\iota_2(w)) = f(w)$

$\textbf{in}\ (\textbf{fold}_{\textbf{nat}}\ g\, k) \qquad$ *// iter* $f\ k = f^{(k+1)}(1)$

$h(\iota_1()) = (\lambda k.(\mathsf{Succ}\, k));\ \ h(\iota_2(f)) = \lambda k.(iter\, f\ k)$

$\textbf{in}\ ((\textbf{fold}_{\textbf{nat}}\, h\, m)\ n)$

$S^- \approx$ System T over inductive data

Credits on inductive data

- Our approach is fairly standard.
  For example, see these surveys and tutorials:
  - J. Adámek, S. Milius, and L.S. Moss, *Initial Algebras and Terminal Coalgebras: a Survey* (2010) draft.
    www.tu-braunschweig.de/Medien-DB/iti/survey_full.pdf
  - J. Gibbons, "Calculating Functional Programs," in *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, LNCS 2297, Springer (2002) 151–203.
  - J. Rutten, "Universal coalgebra: A theory of systems," *Theoretical Computer Science* **249** (2000) 3–80.
- However, we do *not* use initial F-algebras.
  (Ramification breaks them.)

"Classical" formalism #2: $S = S^- +$ coinductive data

Coinductive data declarations: **codata** $\tau = \nu t.\sigma$

Examples

- **codata** $\mathbf{Seq}_\tau = \nu t.(\tau \times t)$
  - Alt: **codata** $\mathbf{Seq}_\tau = \widehat{\mathsf{Constr}}_\tau$ **of** $\tau \times \mathbf{Seq}_\tau$
  - Elms: infinite lists of τ's
- **codata** $\mathbf{Seq}'_\tau = \nu t.(\mathbf{unit} + \tau \times t)$
  - Alt: **codata** $\mathbf{Seq}'_\tau = \widehat{\mathsf{Null}}'_\tau$ **of** **unit** $[\!]$ $\widehat{\mathsf{Constr}}'_\tau$ **of** $\tau \times \mathbf{Seq}'_\tau$
  - Elms: infinite *and finite* lists of τ's
- **codata** $\mathbf{tree}_\tau = \nu\sigma.(\tau \times \sigma \times \sigma)$
  - Elms: Infinite trees with τ labels
- *Computations (traces)*

"Classical" formalism #2: $S = S^- +$ coinductive data

Coinductive data declarations: **codata** $\tau = \nu t.\sigma$

Details

- σ a simple type over t, **unit**, and previously defined base types
- τ's signature functor: $F_\tau t = \sigma$
- The declaration introduces:
  - a constructor $\hat{c}_\tau : F_\tau\tau \to \tau$ — $\hat{c}_\tau$ is a lazy constructor!
  - a destructor $\hat{d}_\tau : \tau \to F_\tau$ — $\hat{d}_\tau$ forces $\hat{c}_\tau$-expressions.
  - a co-recursor $\mathbf{unfold}_\tau : (\forall\sigma)[(\sigma \to F_\tau\sigma) \to \sigma \to \tau]$.
- Semantics of τ: a *largest* set X with $d_\tau : X \cong F_\tau X$.
- $\mathbf{unfold}_\tau$ = τ-structural corecursion = τ-primitive corecursion:
$$\hat{d}_\tau \circ (\mathbf{unfold}_\tau\, f) = F_\tau(\mathbf{unfold}_\tau\, f) \circ f.$$

"Classical" formalism #2: $S = S^- +$ coinductive data

Our running example: **codata** $\mathbf{Seq}_{\mathbf{nat}} = \nu t.(\mathbf{nat} \times t)$ $\equiv \mathbf{nat}^\omega$

$\hat{d} = \hat{d}_{\mathbf{Seq}_{\mathbf{nat}}}$ $\quad \hat{c} = \hat{c}_{\mathbf{Seq}_{\mathbf{nat}}}$ $\quad \mathbf{Seq}_{\mathbf{nat}} \underset{\hat{c}}{\overset{\hat{d}}{\rightleftarrows}} \mathbf{nat} \times \mathbf{Seq}_{\mathbf{nat}}$ $\quad (\hat{d})^{-1} = \hat{c}$.

Unpacking: $\hat{d} \circ (\mathbf{unfold}_{\mathbf{Seq}_{\mathbf{nat}}}\, f) = F_{\mathbf{Seq}_{\mathbf{nat}}}(\mathbf{unfold}_{\mathbf{Seq}_{\mathbf{nat}}}\, f) \circ f$

$\Downarrow$ $\qquad f : \sigma \to \mathbf{nat} \times \sigma$

$\mathbf{unfold}_{\mathbf{Seq}_{\mathbf{nat}}}\, f = \hat{c} \circ F_{\mathbf{Seq}_{\mathbf{nat}}}(\mathbf{unfold}_{\mathbf{Seq}_{\mathbf{nat}}}\, f) \circ f.$ $\qquad \sigma \approx$ type of seeds

$= \hat{c} \circ (id_{\mathbf{nat}^S} \times (\mathbf{unfold}_{\mathbf{Seq}_{\mathbf{nat}}}\, f)) \circ f.$

$\Downarrow$

$\mathbf{unfold}_{\mathbf{Seq}_{\mathbf{nat}}}\, f\, s = \hat{c}\big(((id_{\mathbf{nat}^S} \times (\mathbf{unfold}_{\mathbf{Seq}_{\mathbf{nat}}}\, f)) \circ f)\, s\big).$ $\qquad s \equiv$ a seed

$\approx \hat{c}\big(\,(n, \mathbf{unfold}_{\mathbf{Seq}_{\mathbf{nat}}}\, f\, s')\,\big)$

where $f(s) = (n, s')$.

"Classical" formalism #2: $S = S^- +$ coinductive data

$\mathbf{unfold}_{\mathbf{Seq}_{\mathbf{nat}}}\, f\, s \;\approx\; \underbrace{\hat{c}(}_{\text{lazy}}\; (n, \mathbf{unfold}_{\mathbf{Seq}_{\mathbf{nat}}}\, f\, s')\,)$, where $f(s) = (n, s')$.

Abbreviation: $n :: xs \;\approx\; \hat{c}(\,(n, xs)\,)$

Example: $pos = 1 :: 2 :: 3 :: 4 :: 5 :: \ldots$

$$\begin{aligned} pos &= \mathbf{unfold}_{\mathbf{Seq}_{\mathbf{nat}}} \overbrace{\lambda s.(s, s+1)}^{f}\; 1 \\ &= 1 :: (\mathbf{unfold}_{\mathbf{Seq}_{\mathbf{nat}}}\, f\;\; 2) \\ &= 1 :: 2 :: (\mathbf{unfold}_{\mathbf{Seq}_{\mathbf{nat}}}\, f\;\; 3) \\ &= 1 :: 2 :: 3 :: (\mathbf{unfold}_{\mathbf{Seq}_{\mathbf{nat}}}\, f\;\; 4) \\ &\;\;\vdots \end{aligned}$$

"Classical" formalism #2: $S = S^- +$ coinductive data

$$head\,(n_1 :: n_2 :: \dots) = n_1 \qquad tail\,(n_1 :: n_2 :: \dots) = (n_2 :: \dots)$$

Example: $everyOther\,(x_0 :: x_1 :: x_2 :: \dots) = x_0 :: x_2 :: x_4 :: \dots$

$$xs = x_0 :: x_1 :: x_2 :: x_3 :: x_4 :: x_5 :: \dots$$

$$\begin{aligned} everyOther\ xs &= \mathbf{unfold}_{\mathbf{Seq_{nat}}}\ \overbrace{(\lambda ys\,.\,(head\,ys, tail(tail\,ys)))}^{f}\ xs \\ &= x_0 :: \mathbf{unfold}_{\mathbf{Seq_{nat}}}\ f\ \ (x_2 :: x_3 :: x_4 :: \dots) \\ &= x_0 :: x_2 :: \mathbf{unfold}_{\mathbf{Seq_{nat}}}\ f\ \ (x_4 :: x_5 :: x_6 :: \dots) \\ &= x_0 :: x_2 :: x_4 :: \mathbf{unfold}_{\mathbf{Seq_{nat}}}\ f\ \ (x_6 :: x_7 :: x_8 :: \dots) \end{aligned}$$

"Classical" formalism #2: $S = S^- +$ coinductive data

$$pos = 1 :: 2 :: 3 :: 4 :: 5 :: \dots$$
$$everyOther\,(x_0 :: x_1 :: x_2 :: \dots) = x_0 :: x_2 :: x_4 :: \dots$$

Example: $powers \approx 2^0 :: 2^1 :: 2^2 :: 2^3 :: 2^4 :: \dots$ via a sieve

$$\begin{aligned} powers &= \mathbf{unfold}_{\mathbf{Seq_{nat}}}\ \overbrace{(\lambda ys\,.\,(head\,ys, everyOther(tail\,ys)))}^{f}\ pos \\ &= 1 :: \mathbf{unfold}_{\mathbf{Seq_{nat}}}\ f\ (2 :: 4 :: 6 :: 8 :: 10 :: \dots) \\ &= 1 :: 2 :: \mathbf{unfold}_{\mathbf{Seq_{nat}}}\ f\ (4 :: 8 :: 12 :: 16 :: \dots) \\ &= 1 :: 2 :: 4 :: \mathbf{unfold}_{\mathbf{Seq_{nat}}}\ f\ (8 :: 16 :: 24 :: 32 :: \dots) \\ &= 1 :: 2 :: 4 :: 8 :: \mathbf{unfold}_{\mathbf{Seq_{nat}}}\ f\ (16 :: 32 :: 48 :: 64 :: \dots) \end{aligned}$$

$S \approx$ System T + inductive and conductive data

fold vs. unfold

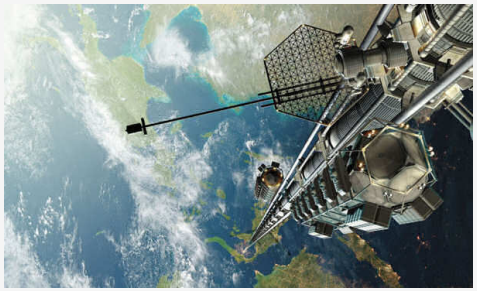
Credits on coinductive data

- Our approach to this is also standard.
  For example, see these (same) surveys and tutorials:
  - J. Adámek, S. Milius, and L.S. Moss, *Initial Algebras and Terminal Coalgebras: a Survey* (2010) draft.
    www.tu-braunschweig.de/Medien-DB/iti/survey_full.pdf
  - J. Gibbons, "Calculating Functional Programs," in *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, LNCS 2297, Springer (2002) 151–203.
  - J. Rutten, "Universal coalgebra: A theory of systems," *Theoretical Computer Science* **249** (2000) 3–80.
- However, we do *not* use final F-coalgebras.
  (Ramification breaks them too.)
- Larry Moss tells us that we (D&R) may be the first ones to write down S.

Problem

- On our modest foundations

- S^- and S give us this



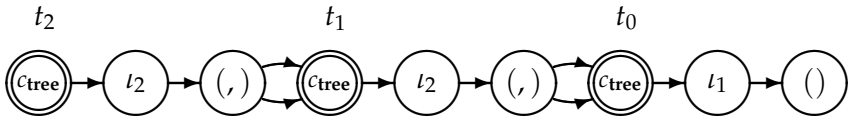
- But we really want something like this

Step 1 to a solution: Low level details about S^-

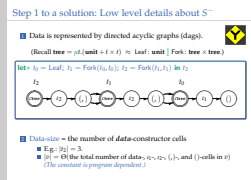
1. Data is represented by directed acyclic graphs (dags).

(Recall **tree** $= \mu t.(\mathbf{unit} + t \times t) \approx \mathsf{Leaf} : \mathbf{unit} \,[\!]\, \mathsf{Fork} : \mathbf{tree} \times \mathbf{tree}$.)

$\mathbf{let}* \; t_0 = \mathsf{Leaf};\; t_1 = \mathsf{Fork}(t_0, t_0);\; t_2 = \mathsf{Fork}(t_1, t_1) \; \mathbf{in} \; t_2$



2. Data-size = the number of ***data***-constructor cells
   - E.g.: $|t_2| = 3$.
   - $|v| = \Theta$(the total number of data-, ι_1-, ι_2-, (,)-, and ()-cells in v)
     (The constant is program dependent.)



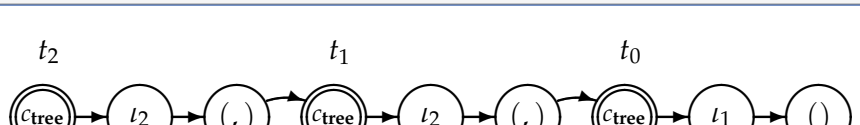
- Our choice may be *ad hoc*, but it is a really popular *ad hoc* choice.
- Pola project: M. Burrell, R. Cockett, and B. Redmond
- Pola: in $\mathsf{Fork}(a, b)$ no sharing between a and b
  - "one-use" not enough
  - Spacial logic: $*$ and $-\!\!*$

Step 1 to a solution: Low level details about S^-

3. Evaluating **fold**-recursions: a dynamic programming problem

   Dynamic programming = *sensible* structural recursion on dags.

   So in a $\mathbf{fold}_{\mathbf{tree}}$-recursion on t_2,
   there are three steps to the recursion, not seven!!!



(See reference to U. Dal Lago, S. Martini, and M. Zorzi (2010) later on.)

Step 1 to a solution: Low level details about S^-

4. The cost of an S^- computation:
   - The evaluation semantics for S^- are given by a collection of structural operational semantics rules. E.g.:

$$\textit{Val:}\ \frac{}{v\theta \downarrow v\theta}\ \begin{pmatrix} v\theta \text{ is a} \\ \text{value} \end{pmatrix} \qquad \textit{Env:}\ \frac{}{x\theta \downarrow v\theta'}\ (\theta(x) = v\theta')$$

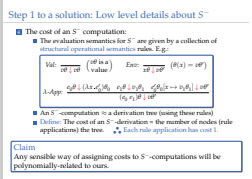
$$\lambda\textit{-App:}\ \frac{e_0\theta \downarrow (\lambda x.e_0')\theta_0 \quad e_1\theta \downarrow v_1\theta_1 \quad e_0'\theta_0[x \mapsto v_1\theta_1] \downarrow v\theta'}{(e_0\ e_1)\theta \downarrow v\theta'}$$

   - An S^--computation $\approx$ a derivation tree (using these rules)
   - Define: The cost of an S^--derivation = the number of nodes (rule applications) the tree. ∴ Each rule application has cost 1.

Claim
Any sensible way of assigning costs to S^--computations will be polynomially-related to ours.

Ramified Structural Recursion and Corecursion

└Step 1 to a solution: Low level details about S^-



- S^- and S will be our "universal" models of computation + cost.
- Not Turing complete, but that is not really a problem.
- Shares the DP approach of evaluating **fold**'s.

Step 2 to a solution: Ramify the data-types

- Why ramify?
  To break vicious circles, e.g., huge recursions (& corecursions).
- What flavor of ramification?

  Normal/Safe based on Bellantoni and Cook's *BC* formalism (***not*** *B*!!) and Leivant's 1995 formalism.
- **data** $\tau = \mu t.\sigma$ introduces
  - the normal type τ with $c_\tau : F_\tau\tau \to \tau$ and $d_\tau : \tau \to F_\tau\tau$ as before.
  - the safe type τ^{S} with $c_{\tau^{\mathsf{S}}} : (F_\tau\tau)^{\mathsf{S}} \to \tau^{\mathsf{S}}$ and $d_{\tau^{\mathsf{S}}}\ \tau^{\mathsf{S}} \to (F_\tau\tau)^{\mathsf{S}}$.
    $((\sigma \times \xi)^{\mathsf{S}} = \sigma^{\mathsf{S}} \times \xi^{\mathsf{S}}.\quad (\sigma + \xi)^{\mathsf{S}} = \sigma^{\mathsf{S}} + \xi^{\mathsf{S}}.\quad ()^{\mathsf{S}} = ().)$
  - $\mathbf{fold}^{\mathsf{S}}_\tau : (\forall\sigma \mid \sigma \text{ is safe})[(F_\tau\sigma \to \sigma) \to \tau \to \sigma]$ Replaces $\mathbf{fold}_\tau$.
    ($\mathbf{fold}^{\mathsf{S}}_\tau$ and $\mathbf{fold}_\tau$: different typing, but the same op. semantics.)

N.B. The normal/safe distinction applies to just ground (level 0) types.

Step 2 to a solution: Ramify the data-types

normal ≈ values that can drive a recursion
safe ≈ values resulting from a recursion

Examples (with just a little sugar)

$plus : \mathbf{nat} \to \mathbf{nat}^{\mathsf{S}} \to \mathbf{nat}^{\mathsf{S}}$

$plus\ x\ y = \mathbf{let}\ f(\iota_1\ w) = y;\ f(\iota_2\ w) = (\mathsf{Succ}^{\mathsf{S}}\ w)\ \mathbf{in}\ (\mathbf{fold}^{\mathsf{S}}_{\mathbf{nat}}\ f\ x)$

$times : \mathbf{nat} \to \mathbf{nat} \to \mathbf{nat}^{\mathsf{S}}$

$times\ x\ y = \mathbf{let}\ f(\iota_1\ w) = \mathsf{Zero}^{\mathsf{S}};\ f(\iota_2\ w) = (plus\ x\ w)\ \mathbf{in}\ (\mathbf{fold}^{\mathsf{S}}_{\mathbf{nat}}\ f\ y)$

$sumLst : \mathbf{natLst} \to \mathbf{nat}^{\mathsf{S}}$

$sumLst\ xs = \mathbf{let}\ g(\iota_1\ w) = \mathsf{Zero}^{\mathsf{S}};\ g(\iota_2(x,t)) = (plus\ x\ t)\ \mathbf{in}\ (\mathbf{fold}^{\mathsf{S}}_{\mathbf{natLst}}\ g\ xs)$

Non-Example

wrong type!

$cube = \lambda x.(times\ x\ \overbrace{(times\ x\ x)}\ )$

Step 2 to a solution: Ramify the data-types

$$\textit{Up-I:}\ \frac{\Gamma \vdash e : \tau}{\Gamma \vdash (\mathbf{up}\ e) : \tau^{\mathsf{S}}}\ (\dagger) \qquad \textit{Down-I:}\ \frac{\Gamma \vdash e : \tau^{\mathsf{S}}}{\Gamma \vdash (\mathbf{down}\ e) : \tau}\ (\star)$$

($\dagger$) τ is a normal base type.

($\star$) ($\dagger$) & each $x \in freeVars(e)$ occurs in a normal-type subterm of e.

- $(\mathbf{up}\ v)$ = a safe-version of v
- $(\mathbf{down}\ v)$ = a normal-version of v

Examples

$cube : \mathbf{nat} \to \mathbf{nat}^{\mathsf{S}}$

$cube = \lambda x.(times\ x\ (\mathbf{down}\ (times\ x\ x)))$

$cube' : \mathbf{nat} \to \mathbf{nat}$

$cube' = \lambda x.(\mathbf{down}(cube\ x))$

- *Down-I* is a λ-calculus adaptation of Bellantoni and Cook's raising rule.
- The raising rule ≈ a specialization of Whitehead and Russell's axiom of reducibility.

Really?

RS_1^-: The ramified version of S^-

$RS_1^- = S^-$ with normal/safe ramified data types
- + **up** and **down**
- + **case**-expressions, +-types, and ×-types restricted to ground level (for simplicity)
- + 2nd-order **fold**^{S}'s (for sanity)

Theorem (RS_1^-: Polynomial-time soundness)

Given an RS_1^- term $x_1 : \gamma_1, \ldots, x_k : \gamma_k \vdash e : \gamma_0$ *where each γ_i is normal or safe, one can construct a polynomial p over $\{\, |x_i| \mid \gamma_i \text{ is normal} \,\}$ such that:*

$$\textit{evaluation-cost}(e\theta) \le p\theta, \quad \textit{for each variable environment } \theta.$$

N.B. *e may contain subterms of arbitrarily high type levels.*

Notes on the proof

RS_1^-: The ramified version of S^-

RS_1^- and incompleteness

Q: Can RS_1^- compute the depth of a **tree**?
What is the problem?
How to compute the max of the depth of two branches?

Q: For **nat**-labeled trees:
Can RS_1^- test whether such a tree has a repeated label?
Why feasible? Distinct labels $\Longrightarrow$ distinct nodes

- ➤ For branching data: We suspect RS_1^- is incomplete.
- ➤ For non-branching data: We strongly suspect RS_1^- *is* complete
  since representations are unique.

Q: How to fix incompleteness? *(Later)*

Sample credits related to RS_1^- (*Very incomplete*)

- S. Bellantoni and S. Cook, "A new recursion-theoretic characterization of the polytime functions," *Computational Complexity* **2** (1992) 97–110.
- D. Leivant, "Ramified recurrence and computational complexity I: Word recurrence and poly-time," *Feasible Mathematics II*, Birkhäuser (1995) 320–343.
- U. Dal Lago, S. Martini, and M. Zorzi, "General Ramified Recurrence is Sound for Polynomial Time," *Electronic Proceedings in Theoretical Computer Science* **23** (2010) 47–62.
- M. Burrell, R. Cockett, and B. Redmond, "Safe recursion revisited I: Categorical semantics for lower complexity," **TCS** (2013) `http://dx.doi.org/10.1016/j.tcs.2013.09.034`

⋮

- N. Danner and J. Royer, "Adventures in time and space," *Logical Methods in Computer Science* **3** (2007) 1–53.

Step 3: Ramify the codata types

Declaring **codata** $\tau = \nu t.\sigma$ introduces

- the normal type τ with $\hat{c}_\tau : F_\tau\tau \to \tau$ and $\widehat{d}_\tau : \tau \to F_\tau\tau$ as before.
- the safe type τ^S with $\hat{c}_{\tau^S} : (F_\tau\tau)^S \to \tau^S$ and $\hat{d}_{\tau^S}\ \tau^S \to (F_\tau\tau)^S$.
- **unfold**$^S_\tau$: $(\forall \sigma \mid \sigma \text{ is safe})[(\sigma \to F_\tau\sigma) \to \sigma \to \boxed{\tau^S}]$ Replaces **unfold**$_\tau$.
  (**unfold**$^S_\tau$ and **unfold**$_\tau$: different typing, but the same op. semantics.)

!! The τ^S in the typing of **unfold**$^S_\tau$ is restrictive trouble.
But τ in place of τ^S leads to infeasibility.

!! And there are other troubles . . .

Step 3: Ramify the codata types

As things stand, the following are allowed:

✔ $pos\colon \mathbf{Seq}^{\mathsf{S}}_{\mathbf{nat}}$ // $\approx [1,2,3,4,\ldots]$

$pos = \mathbf{unfold}^{\mathsf{S}}_{\mathbf{Seq_{nat}}}\ (\lambda k.(k, \mathsf{Succ}^{\mathsf{S}}\ k))\ (\mathsf{Succ}^{\mathsf{S}}\ \mathsf{Zero}^{\mathsf{S}})$

? $everyOther\colon \mathbf{Seq}^{\mathsf{S}}_{\mathbf{nat}} \to \mathbf{Seq}^{\mathsf{S}}_{\mathbf{nat}}$ // $[x_0,x_1,x_2,\ldots] \mapsto [x_0,x_2,x_4,\ldots]$

$everyOther = \lambda xs.\ \mathbf{unfold}^{\mathsf{S}}_{\mathbf{Seq_{nat}}}\ (\lambda ys.(head\ ys, \underbrace{tail(tail\ ys)}_{(\star)}))\ xs$

✘ $powers\colon \mathbf{Seq}^{\mathsf{S}}_{\mathbf{nat}}$ // $\approx [2^0,2^1,2^2,2^3,\ldots]$

$powers = \mathbf{unfold}^{\mathsf{S}}_{\mathbf{Seq_{nat}}}\ (\lambda ys.(head\ ys, everyOther(tail\ ys)))\ pos$

($\star$) Nested $\hat{d}_{\mathbf{Seq}^{\mathsf{S}}_{\mathbf{nat}}}$'s $\rightsquigarrow$ stream speed-ups

Step 3: Ramify the codata types

Definition

(a) ($\mathbf{unfold}^{\mathsf{S}}_{\sigma}$ *step seed*) is speeding if *step* (in n.f.) contains a nested application of safe-codata destructors.

(b) A stepwise expression is one without any speeding $\mathbf{unfold}^{\mathsf{S}}$'s.

The stepwise side-condition for $\mathbf{unfold}^{\mathsf{S}}$

$$\mathbf{unfold}^{\mathsf{S}}_{\tau}\text{-}I\colon\ \frac{\Gamma \vdash f\colon \sigma \to F_\tau\sigma \quad \Gamma \vdash e\colon \sigma}{\Gamma \vdash \mathbf{unfold}^{\mathsf{S}}_{\tau} f\ e\colon \tau^{\mathsf{S}}} \quad (\star)$$

($\star$) τ is a normal codata type with signature functor F_τ, σ is safe, and f is stepwise.

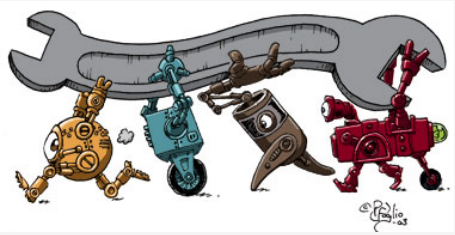
Step 3: Ramify the codata types

Safe-streams are reasonably powerful: *Thunk parades*

$steps_i\colon \mathbf{Seq}^{\mathsf{S}}_{\mathbf{nat}} \to \mathbf{Seq}^{\mathsf{S}}_{\mathbf{nat}}$ // $steps_i[\ldots,x_k,\ldots] = [\ldots, x_k + \binom{k}{i}, \ldots]$

$steps_0\ xs = \mathbf{unfold}^{\mathsf{S}}_{\mathbf{Seq_{nat}}}\ \left((\mathsf{Succ}^{\mathsf{S}} \times id_{\mathbf{Seq}^{\mathsf{S}}_{\mathbf{nat}}}) \circ \hat{d}_{\mathbf{Seq}^{\mathsf{S}}_{\mathbf{nat}}}\right)\ xs$

$steps_{i+1}\ xs = \mathbf{unfold}^{\mathsf{S}}_{\mathbf{Seq_{nat}}}\ \left((id_{\mathbf{nat}^{\mathsf{S}}} \times \boxed{steps_i}) \circ \hat{d}_{\mathbf{Seq}^{\mathsf{S}}_{\mathbf{nat}}}\right)\ xs$



Details

But ...

since codata are lazy, to reach far into a codatum one still needs a $\mathbf{fold}^{\mathsf{S}}$ driven by a normal-datum.

RS_1: The ramified version of S

$RS_1 = S$ with normal/safe ramified data types

$+\ RS_1^-$'s changes

+ the stepwise side-condition on (2^{nd}-order) $\mathbf{unfold}^{\mathsf{S}}$'s

Theorem (RS_1: Polynomial-time soundness)

Given $\boxed{x_1:\gamma_1,\ldots,x_k:\gamma_k \vdash_{RS_1^-} e:\gamma_0}$ *where each* γ_i *is normal or safe,*

we can construct a poly p *over* $\{\, |x_i|, \boxed{(\!|x_i|\!)} \mid \gamma_i \text{ is normal} \,\}$ *such that:*

$$evaluation\text{-}cost(e\theta) \le p\theta, \quad \text{for each variable environment } \theta.$$

$(\!|x|\!)$ = the codata size of $x \approx$ Kapron-Cook 1st-order size

RS_1: The ramified version of S

RS_1 and incompleteness: Normal maps are missing.

- E.g.:

$$map\colon (\mathbf{nat} \to \mathbf{nat}) \to \mathbf{Seq}_{\mathbf{nat}} \to \mathbf{Seq}_{\mathbf{nat}}$$
$$map\, f\, [\ldots, n_k, \ldots] = [\ldots, f\, n_k, \ldots]$$

- These are unproblematically feasible, but ...
- RS_1 cannot define them.

Q: Normal-maps + RS_1 = a kind of completeness?

Credits related to RS_1

- M. Burrell, R. Cockett, and B. Redmond, "Safe recursion revisited I: Categorical semantics for lower complexity," **TCS** (2013) `http://dx.doi.org/10.1016/j.tcs.2013.09.034`
- H. Férée, E. Hainry, M. Hoyrup, and R. Péchoux, "Interpretation of stream programs: Characterizing type 2 polynomial time complexity," *Algorithms and Computation*, Springer LNCS 6506 (2010) 291–303.
- R. Ramyaa and D. Leivant, "Feasible functions over co-inductive data," *Logic, Language, Information and Computation*, Springer LNCS 6188 (2010) 191–203.
- R. Ramyaa and D. Leivant, "Ramified corecurrence and logspace," *MFPS XXVII, ENTCS 276* (2011) 247–261.

So what does our solution (RS_1) solve?

RS_1

= S
- + normal/safe ramified types
- + dags/DP-**fold**^{S}'s
- + **up** and **down**
- + stepwise **unfold**^{S}'s
- + 2nd-order **fold**^{S}'s & **unfold**^{S}'s

- polytime sound
- incomplete over codata
- likely incomplete over data

Re: feasible computation over data & codata (1st order)

- It gives an uncluttered look at the territory.
- It lets us compute quite a lot.
- It exposes some clear problems.
- It's soundness proofs provide analysis tools. *(Not in this talk.)*
- It provides a platform for further exploration, either
  - to build on
  - or to reject.

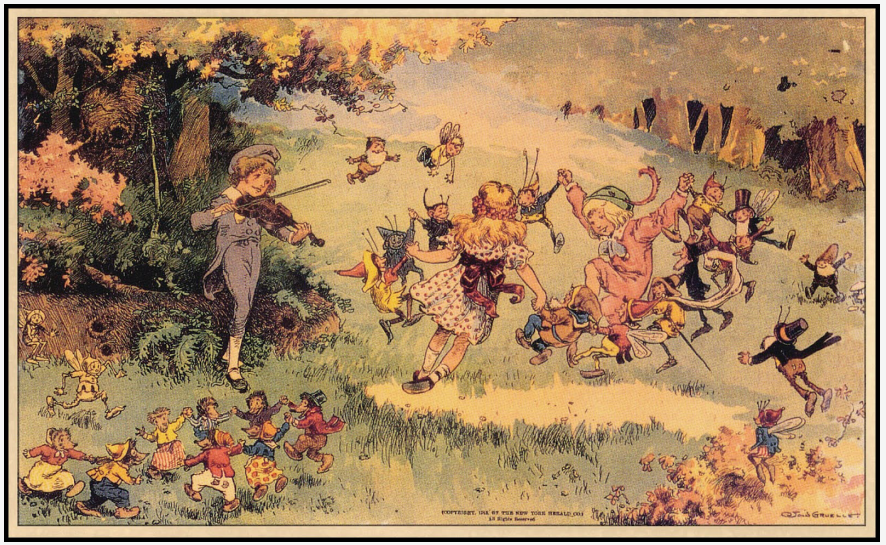
Some specific open problems

- replacing and/or supplementing **fold**^{S}
  (tree-compressions, but higher-rank data = trouble)
- **unfold**^{S} + normal-maps over codata = **??**
- fancier notions of data and codata / higher-order **fold**^{S}'s and **unfold**^{S}'s
- restrict types and lazy data. E.g.,
  - Restricting to **nat** + 0-1-streams yields logspace stream functions
  - restricting to **nat** + lazy 0-1-strings would yield logspace functions
- algebraic/categorical foundations of data and codata
  - Basis for Bird-style program transformations for optimizations.
  - Broken by ramification
  - Do how things break tell us something? turn What is recoverable? How does it tie to optimization?

Some specific open problems
- replacing and/or supplementing fold$^{\mathbf{a}}$ (tree-compressions, but higher-rank data = trouble)
- unfold$^{\mathbf{a}}$ + normal-maps over codata = ??
- fancier notions of data and codata / higher-order fold$^{\mathbf{a}}$'s and unfold$^{\mathbf{a}}$'s
- restrict types and lazy data. E.g.,
  - Restricting to **nat** + 0-1-streams yields logspace stream functions
  - restricting to **nat** + lazy 0-1-strings would yield logspace functions
- algebraic/categorical foundations of data and codata
  - Basis for Bird-style program transformations for optimizations.
  - Broken by ramification
  - Do how things break tell us something? turn What is recoverable? How does it tie to optimization?

- Aehlig-Cook-Nguyen and two sorted complexity classes

Enough! We are done!!



Footnotes

fold and unfold / S^- and S

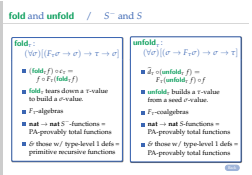
fold$_\tau$:
$(\forall\sigma)[(F_\tau\sigma \to \sigma) \to \tau \to \sigma]$

- $(\mathbf{fold}_\tau f) \circ c_\tau = f \circ F_\tau(\mathbf{fold}_\tau f)$
- **fold**$_\tau$ tears down a τ-value to build a σ-value.
- F_τ-algebras
- **nat** $\to$ **nat** S^--functions = PA-provably total functions
- & those w/ type-level 1 defs = primitive recursive functions

unfold$_\tau$:
$(\forall\sigma)[(\sigma \to F_\tau\sigma) \to \sigma \to \tau]$

- $\widehat{d}_\tau \circ (\mathbf{unfold}_\tau f) = F_\tau(\mathbf{unfold}_\tau f) \circ f$
- **unfold**$_\tau$ builds a τ-value from a seed σ-value.
- F_τ-coalgebras
- **nat** $\to$ **nat** S-functions = PA-provably total functions
- & those w/ type-level 1 defs = PA-provably total functions

Back



- S is one reading of David Turner's "total functional programming." Although it is probably too spare and too ML-ish for him.
- The reason type-level 1 S-functions are so powerful is that, for each ordinal $\alpha < \epsilon_0$, one can use codata to implement a notation system for the ordinals $< \alpha$.

Warning: Making sense of W&R on this stuff is vexing

Bellantoni and Cook, 1992, §5

One further adds the following "Raising" rule: if function $f(\vec{x};)$ of all normal inputs is in the class with safe type output, then the function f^ν is in the class with normal type output defined by $f^\nu(\vec{x};) = f(\vec{x};)$.

Whitehead and Russell, PM 1/e, Vol. 1, 1910, page 174

Let fu be a function, of any order, of an argument u, which may itself be either an individual or a function of any order. If f is a matrix, we write the function of the form $f!u$; in such a case we call f a predicative function.

matrix $\approx$ no (free) apparent variables real/apparent variables

Back

Notes on the proof of RS_1^- poly-time boundedness

- Bellantoni & Cook's poly-max bounds
  $\rightsquigarrow$
  poly-heap bounds (to account for structure sharing)
- to deal with the (internal) higher-types: D&R's time complexity semantics
  - Higher-type terms have two sorts of complexity
  - cost = cost to evaluate the term to a value
  - potential = costs associated with *using* the higher-type value
  - (Also see Sands, Gurr, Shultis, van Stone, ...)

Back

Details for $steps_1$

$$xs = x_0 :: x_1 :: x_2 :: x_3 :: x_4 :: \ldots$$

$$steps_0\, xs = \mathbf{unfold}^{\mathsf{S}}_{\mathbf{Seq_{nat}}} \left((\mathsf{Succ}^{\mathsf{S}} \times id_{\mathbf{Seq}^{\mathsf{S}}_{\mathbf{nat}}}) \circ \widehat{d}_{\mathbf{Seq}^{\mathsf{S}}_{\mathbf{nat}}} \right) xs$$
$$= (x_0 + 1) :: (x_1 + 1) :: (x_2 + 1) :: (x_3 + 1) :: (x_4 + 1) :: \ldots$$

$$steps_1\, xs = \mathbf{unfold}^{\mathsf{S}}_{\mathbf{Seq_{nat}}} \overbrace{\left((id_{\mathbf{nat}^{\mathsf{S}}} \times \boxed{steps_0}) \circ \widehat{d}_{\mathbf{Seq}^{\mathsf{S}}_{\mathbf{nat}}} \right)}^{f} xs$$
$$= x_0 :: \mathbf{unfold}^{\mathsf{S}}_{\mathbf{Seq_{nat}}}\ f\ \ (steps_0(x_1 :: x_2 :: \ldots))$$
$$= x_0 :: (x_1 + 1) :: \mathbf{unfold}^{\mathsf{S}}_{\mathbf{Seq_{nat}}}\ f\ \ (steps_0^{(2)}(x_2 :: x_3 :: \ldots))$$
$$= x_0 :: (x_1 + 1) :: (x_2 + 2) :: \mathbf{unfold}^{\mathsf{S}}_{\mathbf{Seq_{nat}}}\ f\ \ (steps_0^{(3)}(x_3 :: x_4 :: \ldots))$$
$$\vdots$$

Back