Robin Cockett
University of Calgary
CANADA

July 2009

... Categorical Semantics of P-time

Joint work with:
Mike Burrell (Western) and Brian Redmond (Calgary).
1. Introduction
   - P-time
   - Implicit characterizations
2. The logic of safety
   - Basic features
   - Other features ...
3. Data in the logic of safety
   - Modalities
   - Inductive data
   - Example programs
   - Recursion in Pola
4. Categorical semantics
   - Strong polarized categories
   - Bundle fibrations
   - The lift
What is P-time programming?

Standard definition of polynomial time computation (P-time):

A computation which can be completed by a Turing machine in a number of steps bounded by a polynomial in the length of its input tape.
What is P-time?

Advantages of definition:
- Very simple!
- Directly related to standard Turing notion of “computable”.

Disadvantages of definition:
- No one programs using a Turing machine!
  - Machine level view of data.
  - Explicit counts steps.
  - Lacks an independent semantics.
Things we would like to know ... 
... for P-time programming.

- What should the constructs of a P-time language be?
- Can types of P-time programs be inferred?
- What (inductive/coinductive) data should be supported by P-time languages?
- Can a P-time language be expressive?

Old questions ...
Implicit complexity ...

Explicit: Direct counting of computation steps; realized in Sets ...

Implicit: P-time by construction; realized in its own logical setting.

- Need an implicit description ...
- With a clean (categorical) semantics ...
- .... WHICH CAN BE IMPLEMENTED!
POLA: why is it interesting?

- Based on a categorical semantics (functional in style).
- P-time complete: every P-ime function can be expressed and all correctly typed programs are P-time.
- Programming constructs
  - affine types
  - recursion scheme
  - peeking (for P-space)

Prototype should be available soon!
Website (still being developed): pola@wizardlike.ca
... because it is a programming language ...

Colson’s objection:

First-order primitive recursion cannot express algorithms efficiently!

*Is there any hope for P-time?*
From recursion theory ...

1964: Cobham’s function algebra $\mathcal{F}$: uses bounded (by existing function) recursion and an added initial function $x \# y = 2^{|x| \cdot |y|}$ to get things going!

1992: Leivant introduced a notion of predicative recursion using a “stratified” type system (two levels suffice).

1992: Bellantoni and Cook introduce the notion of safe-recursion.

1994: Jim Otto proposes a categorical formulation ...

1998: Martin Hoffman (SLR) used modal logic to express P-time using a Topos theoretic semantics.

1998: A. Asperti modified the above to “Light Affine Logic” capturing the essence of what was needed.

2000: Y. Lafont modified the “bang” ... providing an alternative but similar perspective;

2004: A. Murawski and L. Ong interpret Bellantoni and Cook’s function algebra $\mathcal{B}$ in light affine logic.

*Universal quantification used to introduce arithmetic power ...*
Descriptive complexity ..... 

.... logics in which the functions whose graphs are provably total are exactly the P-time functions (e.g. \( \mu \)-logic with linear order).

(does not help with programming!)
Bellantoni and Cook separated the variables of a function $f$ into two zones:

$$f(\langle \vec{x} \mid \vec{y} \rangle)$$

The “normal variables” and the “safe variables”.

- The class $\mathcal{B}$ is the smallest class of functions containing initial function $(\text{succ}_0, \text{succ}_1, \text{zero})$ and closed under safe composition and safe recursion.

- $f \in \mathcal{B}$ is polynomial time in its normal variables and bounded by linear time in its safe variables.
Bellantoni & Cook on safety

**Safe Composition:** Define the new function $f$ by

$$f(\bar{x}|\bar{y}) = h(\bar{r}(\bar{x}|\bar{y})|\bar{t}(\bar{x}|\bar{y}))$$

where $h$, $\bar{r}$ and $\bar{t}$ are in $\mathcal{B}$.

**Safe Recursion:** Define the new function $f$ by

$$f(\text{zero}, \bar{x} | \bar{y}) = g(\bar{x} | \bar{y})$$
$$f(\text{succ}_i(z), \bar{x} | \bar{y}) = h_i(z, \bar{x} | \bar{y}, f(z, \bar{x} | \bar{y}))$$

where $h_i$ and $g$ are in $\mathcal{B}$.

Notice that the result of the recursive call is substituted in a safe position and that safe composition prevents it from being copied into a normal position.
Hofmann on safety

Martin Hofmann made some key observations on safety:

- If the safe computations are forced to be \textit{constant time} then the proof that safe recursion is contained in P-time is much simpler ...
  (Idea: polynomially iterating a constant time program is always polynomial!)

- BUT to achieve constant time computations in the safe zone one needs an \textit{affine} logic (which neither Leivant nor Bellatoni and Cook had assumed).

\textit{Copying} data is \textit{not} a constant time operation!
Some tricky issues ...

- Cannot have initial (inductive) data in a P-time setting.
- BUT ... to program one wants inductive and coinductive data!
- Binary numbers are not isomorphic to unary numbers ...
- AND trees are just wierd!

Bellatoni and Cook consider only binary numbers. Leivant and Hofmann consider more general inductive data BUT completely disagreed on how such data should behave!
We now introduce the **basic logic of safety** as a categorical proof theory ... 

Proofs are programs ... and are maps in a semantic category/module.

*Proof theory*  ⇔  *Categorical doctrine*

This delivers a term logic, a reduction system, and a categorical semantics ...
Basic logic of safety

The basic logic of safety has three sorts of sequents:

\[ \Gamma \vdash_o X \quad \Gamma \mid \Delta \leftrightarrow_{op} Y \quad \Delta \vdash_p Y \]

**Opponent (or normal) sequent:** Antecedent, \( \Gamma \), contains opponent (or normal) types. Conclusion, \( X \), is an opponent type.

**Cross sequent:** Antecedent split into two zones: opponent types, \( \Gamma \), followed by player types, \( \Delta \). Conclusion, \( Y \), is a player type.

**Player (or safe) sequent:** Antecedent, \( \Delta \), contains player (or safe) types. Conclusion, \( Y \), is player type.

A polarized logic: safe = player, ordinary = opponent.
Introduction
The logic of safety
Data in the logic of safety
Categorical semantics

Basic logic of safety

Identity sequents, weakening and contraction

Note: can weaken in all contexts but only o-context enjoys contraction.

Robin Cockett  University of Calgary  CANADA  FICS 2009: POLA for PTIME programming
The basic logic of safety, cont’d

The logic has six cuts:

\[
\begin{align*}
\Gamma \vdash_0 t : X & \quad \Gamma, x : X \vdash t' : X' \quad \text{cut}_0 \\
\Gamma \vdash_0 t'[x/t] : X' & \\
\Delta_1 \vdash p_1 s : Y & \quad \Delta_2, y : Y \vdash p_2 s' : Y' \\
\Delta_1, \Delta_2 \vdash p_2 [s/y] : Y' & \quad \text{cut}_p \\
\Delta_1 \vdash p_1 s : Y & \quad \Gamma | \Delta_2, y : Y \rightarrow s' : Y' \\
\Gamma | \Delta_1, \Delta_2 \rightarrow s'[s/y] : Y' & \quad \text{cut}_{p,op} \\
\Gamma \vdash_0 t : X & \quad \Gamma, x : X | \Delta \equiv s : Y \\
\Gamma | \Delta \equiv s[t/x] : Y & \quad \text{cut}_{o,op} \\
\Gamma | \Delta_1 \equiv s : Y & \quad \Delta_2, y : Y \vdash p s' : Y' \\
\Gamma | \Delta_1, \Delta_2 \equiv s'[s/y] : Y' & \quad \text{cut}_{op,p} \\
\Gamma | \Delta_1 \equiv s : Y & \quad \Gamma | \Delta_2, y : Y \rightarrow s' : Y' \\
\Gamma | \Delta_1, \Delta_2 \equiv s'[s/y] : Y' & \quad \text{cut}_{op,op}
\end{align*}
\]

The cut rules (explicit substitution)
The basic logic of safety, cont’d

Lifting is functorial:

\[ \begin{align*}
\Delta \vdash p \quad s : Y \\
\Delta \quad \Delta \mapsto s : Y \\
\uparrow(\Delta) \quad \Delta \mapsto s : Y \\
\uparrow(\Delta) \quad \uparrow(s) : \uparrow(Y)
\end{align*} \]}
The basic logic of safety, cont’d

\[ \Gamma \vdash_\circ t : X \quad \frac{\Delta \vdash p s : Y}{\Delta, () : 1 \vdash_\circ t : X} \]

\[ \frac{\Gamma, () : 1 \vdash_\circ t : X}{\Gamma \vdash_\circ () : 1} \quad \frac{\Delta, () : 1 \vdash p s : Y}{\Delta \vdash p () : 1} \]

\[ \Gamma \mid \Delta \mapsto () : 1 \]

\[ \frac{\Gamma, x_1 : X_1, x_2 : X_2 \vdash_\circ t : X}{\Gamma, (x_1, x_2) : X_1 \times X_2 \vdash_\circ t : X} \quad \frac{\Delta, y_1 : Y_1, y_2 : Y_2 \vdash p s : Y}{\Delta, (y_1, y_2) : Y_1 \otimes Y_2 \vdash p s : Y} \]

\[ \frac{\Gamma, x_1 : X_1, x_2 : X_2 \mid \Delta \mapsto t : X}{\Gamma, (x_1, x_2) : X_1 \times X_2 \mid \Delta \mapsto t : X} \quad \frac{\Gamma \mid \Delta, y_1 : Y_1, y_2 : Y_2 \mapsto s : Y}{\Gamma \mid \Delta, (y_1, y_2) : Y_1 \otimes Y_2 \mapsto s : Y} \]

\[ \frac{\Gamma \vdash_\circ t_1 : X_1 \quad \Gamma \vdash_\circ t_2 : X_2}{\Gamma \vdash_\circ (t_1, t_2) : X_1 \times X_2} \quad \frac{\Delta_1 \vdash p s_1 : Y_1 \quad \Delta_2 \vdash p s_2 : Y_2}{\Delta_1, \Delta_2 \vdash p (s_1, s_2) : Y_1 \otimes Y_2} \]

\[ \Gamma \mid \Delta_1 \mapsto s_1 : Y_1 \quad \Gamma \mid \Delta_2 \mapsto s_2 : Y_2 \]

\[ \Gamma \mid \Delta_1, \Delta_2 \mapsto \langle s_1, s_2 \rangle : Y_1 \otimes Y_2 \]

Structural product/tensor judgements
The basic logic of safety, cont’d

The following are naturally isomorphic:

\[ \uparrow (Y_1 \otimes Y_2) \cong \uparrow (Y_1 \times Y_2) \quad \text{and} \quad \uparrow (1) \cong 1 \]

Proof of 1:

\[
\begin{align*}
Y_1 \vdash_p Y_1 & \quad Y_2 \vdash_p Y_2 \\
Y_1 \otimes Y_2 \vdash_p Y_1 & \quad Y_1 \otimes Y_2 \vdash_p Y_2 \\
\uparrow (Y_1 \otimes Y_2) \vdash_o \uparrow (Y_1) & \quad \uparrow (Y_1 \otimes Y_2) \vdash_o \uparrow (Y_2) \\
\uparrow (Y_1 \otimes Y_2) \vdash_o \uparrow (Y_1 \times Y_2) & \quad \uparrow (Y_1 \otimes Y_2) \vdash_o \uparrow (Y_1 \otimes Y_2)
\end{align*}
\]
The logic of safety

Data in the logic of safety

Categorical semantics

Cut-elimination

Lemma

The basic logic of safety enjoys cut elimination.

The proof uses a system of rewrite rules:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y[t/x]$</td>
<td>$\Rightarrow y$ ( y \neq x ), $x$ and $y$ variables</td>
</tr>
<tr>
<td>$x[t/x]$</td>
<td>$\Rightarrow t$</td>
</tr>
<tr>
<td>$s[\uparrow(t)/\uparrow(p)]$</td>
<td>$\Rightarrow s[t/p]$</td>
</tr>
<tr>
<td>$\uparrow(t')(t/p)$</td>
<td>$\Rightarrow \uparrow(t'[t/p])$</td>
</tr>
<tr>
<td>$()[t/p]$</td>
<td>$\Rightarrow ()$</td>
</tr>
<tr>
<td>$(s_1, s_2)[t/p]$</td>
<td>$\Rightarrow (s_1[t/p], s_2[t/p])$</td>
</tr>
<tr>
<td>$s[()]$</td>
<td>$\Rightarrow s$</td>
</tr>
<tr>
<td>$s[(t_1, t_2)/(p, q)]$</td>
<td>$\Rightarrow (s[t_1/p])[t_2/q] = (s[t_2/q])[t_1/p]$</td>
</tr>
<tr>
<td>$t{p =_r q}[s/r']$</td>
<td>$\Rightarrow t[s/r']{p =_r q}$ ( r \neq r' )</td>
</tr>
<tr>
<td>$t{p =_r q}[s/r]$</td>
<td>$\Rightarrow t[s/p][s/q]$</td>
</tr>
<tr>
<td>$t[s{p =_r q}/y]$</td>
<td>$\Rightarrow t[s/y]{p =_r q}$</td>
</tr>
</tbody>
</table>

Rewrite Rules
One can increase the flexibility of the system by directly adding linear function spaces $A \rightarrow B$ to the p-world. The typing judgements are as follows:

$\Delta_1 \vdash_p Y_1 \quad \Delta_2, Y_2 \vdash_p Y$

$\Delta_1, \Delta_2, Y_1 \rightarrow Y_2 \vdash_p Y$

$\Gamma \mid \Delta_1 \vdash Y_1 \quad \Delta_2, Y_2 \vdash_p Y$

$\Gamma \mid \Delta_1, \Delta_2, Y_1 \rightarrow Y_2 \vdash Y$

$\Delta_1 \vdash_p Y_1 \quad \Gamma \mid \Delta_2, Y_2 \vdash Y$

$\Delta_1 \vdash_p Y_1 \quad \Gamma \mid \Delta_2, Y_2 \rightarrow Y$

$\Delta, Y_1 \vdash_p Y_2$

$\Delta \vdash_p Y_1 \rightarrow Y_2$

$\Gamma \mid \Delta \vdash Y_1 \rightarrow Y_2$

$\Gamma \mid \Delta \vdash Y_1 \rightarrow Y_2$

Judgements for p-function spaces
Coproducts (or “sums”) are described by the following judgements:

\[
\begin{align*}
\Gamma, X_1 \vdash_o X & \quad \Gamma, X_2 \vdash_o X \\
\quad \quad \Gamma, X_1 +_o X_2 \vdash_o X \\
\Gamma, X_1 \mid \Delta \rightsquigarrow Y & \quad \Gamma, X_2 \mid \Delta \rightsquigarrow Y \\
\quad \quad \quad \quad \Gamma, X_1 +_o X_2 \mid \Delta \rightsquigarrow Y \\
\quad \quad \quad \quad \quad \Gamma \vdash_o X_1 \\
\quad \quad \quad \quad \quad \quad \Gamma \vdash_o X_1 +_o X_2 \\
\quad \quad \quad \quad \quad \quad \quad \Delta \vdash_p Y_1 \\
\quad \quad \quad \quad \quad \quad \quad \quad \Delta \vdash_p Y_1 +_p Y_2 \\
\quad \quad \quad \quad \quad \quad \quad \quad \Gamma \mid \Delta \rightsquigarrow Y_1 \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \Gamma \mid \Delta \rightsquigarrow Y_1 +_p Y_2 \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \Gamma, \uparrow(X_1) \mid \Delta \rightsquigarrow Y \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \Gamma, \uparrow(X_2) \mid \Delta \rightsquigarrow Y \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \Gamma, \uparrow(X_1 +_p X_2) \mid \Delta \rightsquigarrow Y \\
\end{align*}
\]

\[
\begin{align*}
\Delta, Y_1 \vdash_p Y & \quad \Delta, Y_2 \vdash_p Y \\
\quad \quad \Delta, Y_1 +_p Y_2 \vdash_p Y \\
\Gamma \mid \Delta, Y_1 \rightsquigarrow Y & \quad \Gamma \mid \Delta, Y_2 \rightsquigarrow Y \\
\quad \quad \quad \quad \Gamma \mid \Delta, Y_1 +_p Y_2 \rightsquigarrow Y \\
\quad \quad \quad \quad \quad \Gamma \vdash_o Y_2 \\
\quad \quad \quad \quad \quad \quad \Gamma \vdash_o Y_2 +_o Y_1 \\
\quad \quad \quad \quad \quad \quad \quad \Delta \vdash_p Y_1 \\
\quad \quad \quad \quad \quad \quad \quad \quad \Delta \vdash_p Y_1 +_p Y_2 \\
\quad \quad \quad \quad \quad \quad \quad \quad \Gamma \mid \Delta \rightsquigarrow Y_2 \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \Gamma \mid \Delta \rightsquigarrow Y_1 +_p Y_2 \\
\end{align*}
\]

Judgements for sums
Sums

Judgements for sum units:

\[
\Gamma, 0 \vdash_\sigma X \\
\Delta, 0 \vdash_\rho Y \\
\Gamma, 0 \mid \Delta \leftrightarrow Y \\
\Gamma \mid \Delta, 0 \leftrightarrow Y
\]

Note: o-products distribute over o-sums, p-tensor distributes over p-sums.
Products for the player

The p-world can have *products* (in addition to the affine tensor) ... crucially products will be evaluated lazily on destruction.

The only way to look at a product is to “destruct” it. Player products are *not* assumed to distribute over p-sums! Peeking and p-space ...
Inductive data in the logic of safety is built from modalities (rather than functors).

Inductive data is functorial in the o-world but are *not* functorial in the p-world ...

(In particular inductive data does not form a modality).

*Things fall apart* ...
A modality is a pair of type constructors $M_p$ and $M_o$ for player and for opponent types such that:

| $\Gamma, X_1 \vdash_o X'_1 \quad \ldots \quad \Gamma, X_n \vdash_o X'_n$ | $\Delta_1, Y_1 \vdash_p Y'_1 \quad \ldots \quad \Delta_n, Y_n \vdash_p Y'_n$ |
| $\Gamma, M_o(X_1, \ldots, X_n) \vdash_o M_o(X'_1, \ldots, X'_n)$ | $\bar{\Delta}, M_p(Y_1, \ldots, Y_n) \vdash_p M_p(Y'_1, \ldots, Y'_n)$ |

| $\Gamma, X_1 \vdash_1 Y_1 \quad \ldots \quad \Gamma, X_n \vdash_1 Y_n$ | $\Gamma, \bar{\Delta} \vdash_p M_p(Y_1, \ldots, Y_n)$ |
| $\Gamma, M_o(X_1, \ldots, X_n) \vdash_1 \bar{\Delta}$ | $\Gamma, M_o(Y_1, \ldots, Y_n) \vdash_p \bar{\Delta}$ |

| $\Gamma \vdash_1 \bar{\Delta}, M_p(Y_1, \ldots, Y_n)$ | $\Gamma, \bar{\Delta} \vdash_p M_p(Y_1, \ldots, Y_n)$ |

| $\Gamma, M_o(\uparrow(Y_1), \ldots, \uparrow(Y_n)) \vdash_o Y$ | $\Gamma, \uparrow(M_p(Y_1, \ldots, Y_n)) \vdash_o Y$ |
| $\Gamma, \uparrow(M_o(Y_1, \ldots, Y_n)) \vdash_1 Y$ | $\Gamma, \uparrow(M_p(Y_1, \ldots, Y_n)) \vdash_1 Y$ |

Judgements for modalities

Functors in both worlds: strong in both worlds.
Modalities

Three key examples:

- Affine tensor modality: $\otimes = (\times, \otimes)$
- Product modality: $\times = (\times, \times)$
- Sum modality: $+ = (+, +)$
There is an isomorphism of modalities:

\[ \uparrow (M_p(Y_1, \ldots, Y_n)) \cong M_o(\uparrow (Y_1), \ldots, \uparrow (Y_n)) \]

Proof:

\[ \begin{align*}
Y_1 & \vdash_p Y_1 & & Y_n \vdash_p Y_n \\
\uparrow (Y_1) & \vdash_o \uparrow (Y_1) & & \ldots & \uparrow (Y_n) & \vdash_o \uparrow (Y_n) \\
M_o(\uparrow (Y_1), \ldots, \uparrow (Y_n)) & \vdash_o M_o(\uparrow (Y_1), \ldots, \uparrow (Y_n)) \\
\uparrow (M_p(Y_1, \ldots, Y_n)) & \vdash_o M_p(Y_1, \ldots, Y_n)
\end{align*} \]
An inductive datatype is delivered in the p-world by a specification of the form:

\[
\text{data } D(A) \rightarrow C = \begin{cases} 
    c_1 : M_1(C, A) & \rightarrow & C \\
    \cdots \\
    c_k : M_k(C, A) & \rightarrow & C 
\end{cases}
\]

in which the \( c_i \) are constructors and \( M_i \) is a modality a list of p-types \( A \) and the p-type variable \( C \) (representing the fixed point).
Data types, cont’d

Example datatypes include:

```
data Nat → C =
    zero : 1 → C
    succ : C → C

data BNat → C =
    z : 1 → C
    s0 : C → C
    s1 : C → C

data List(A) → C =
    nil : 1 → C
    cons : C ⊗ A → C

data HTree(A) → C =
    leaf : A → C
    node : C ⊗ C → C

data LTree(A) → C =
    leaf : A → C
    node : C × C → C
```
Construction makes datatypes fixed points in the p-world (and in the o-world).

\[
\begin{align*}
\{ \Delta, y_i : F_i(D(A), A) \vdash p \ t_i : Y \}_i \\
\Delta, D(A) \vdash p \text{ case } y \text{ of } \begin{cases} 
C_1(y_1).t_1 \\
\vdots \\
C_k(y_k).t_k 
\end{cases} : Y \\
\Delta \vdash p \ t : F_i(D(A), A) \\
\Delta \vdash p \ C_i(t) : D(A) \\
\{ \Gamma \mid y_i : F_i(D(A), A), \Delta \vdash t_i : Y \}_{i=1,\ldots,k} \\
\Gamma \mid y : D(A), \Delta \vdash \text{case } y \text{ of } \begin{cases} 
C_1(y_1).t_1 \\
\vdots \\
C_k(y_k).t_k 
\end{cases} : Y \\
\Gamma \mid \Delta \vdash t : F_i(D(A), A) \\
\Gamma \mid \Delta \vdash C_i(t) : D(A) \\
\end{align*}
\]
Safe recursion is expressed with a “recursion box”:

\[
\begin{array}{c}
\Gamma, \forall C \\
f : | C, \Sigma \leftrightarrow Y \\
\end{array}
\]

\[
\begin{array}{c}
\{ \\
\begin{array}{c}
y_i' : F_i(D(A), A) \mid y_i : F_i(C, A), w : \Sigma \leftrightarrow t_i : Y \\
\end{array} \\
i = 1, \ldots, k
\end{array}
\]

\[\Gamma, z' : D(A) \mid w' : \Sigma \leftrightarrow \text{fold } f(z, \tilde{w}) \text{ as } \begin{array}{c}
C_1(y_1'@y_1) . t_1 \\
\vdots \\
C_k(y_k'@y_k) . t_k
\end{array} \text{ in } f(z', w') : Y\]

Inside the box, we may use the “global” context \(\Gamma\) by the following rules:

\[
\begin{array}{c}
\Gamma_1, X \vdash_o X_1 \\
\Gamma_1 \vdash_o X_1
\end{array}
\]

\[
\begin{array}{c}
\Gamma_1, X \mid \Delta_1 \leftrightarrow Y_1 \\
\Gamma_1 \mid \Delta_1 \leftrightarrow Y_1
\end{array} \quad \text{(provided } X \in \Gamma)\]

Global context \textit{cannot be changed} inside the box...
Basic programs

Addition of natural numbers:

\[
\forall C : f(n, y) : C, N \rightarrow N
\]
\[
\begin{array}{c}
  f(n, y) : C, N \rightarrow N \\
  succ(n) : N \rightarrow N \\
  y : N \rightarrow y : N
\end{array}
\]
\[
\begin{array}{c}
  succ(f(n, y)) : C, N \rightarrow N \\
  add(x, y) : \uparrow(N) | N \rightarrow N
\end{array}
\]

Multiplication of natural numbers:

\[
\forall C : x : \uparrow(N)
\]
\[
\begin{array}{c}
  f(x|n) : C \rightarrow N \\
  add(x'|n) : \uparrow(N) | N \rightarrow N \\
  f(x|n) : C \rightarrow N \\
  add(x|n) : N \rightarrow N
\end{array}
\]
\[
\begin{array}{c}
  (0 : 1 \rightarrow \text{zero} : N) \\
  add(x|f(x|n)) : C \rightarrow N
\end{array}
\]
\[
\begin{array}{c}
  \text{mul}(x, y) : \uparrow(N), \uparrow(N) \rightarrow N
\end{array}
\]
Simple programs, cont’d

Summing a list of (unary) numbers:

\[
\forall C : 
\begin{array}{|c|c|}
\hline
f(|w|) & C \leftrightarrow N \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
f(|w|) : |C \leftrightarrow N & \text{add}(z|y) : \uparrow(N) | N \leftrightarrow N \\
\hline
\end{array}
\]

(0) : 1 \vdash p \text{sfzero}() : N

\[
\begin{array}{|c|c|}
\hline
\text{add}(z|f(|w|)) : \uparrow(N) | C \leftrightarrow N \\
\hline
\end{array}
\]

sumlist(x|) : \uparrow((\text{list})(N)) | \leftrightarrow N

Reversing a list:

\[
\forall C : 
\begin{array}{|c|c|}
\hline
f(|w, []|) : |C, \text{list}(A) \leftrightarrow \text{list}(A) \\
\hline
\end{array}
\]

| C, \text{list}(A) \leftrightarrow \text{list}(A) \quad \text{list}(A), A \vdash p \text{list}(A) \\
| C, \text{list}(A), A \leftrightarrow \text{list}(A) \\
\begin{array}{|c|c|}
\hline
\text{list}(A) \vdash p \text{list}(A_p) & A_0 \vdash C, \text{list}(A) \leftrightarrow \text{list}(A) \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
\text{revlist}(x|[]) : \uparrow((\text{list})(A)) | \text{list}(A) \leftrightarrow \text{list}(A) \\
\hline
\end{array}
\]
The previous definitions may look more familiar as follows:

\[
\begin{align*}
\text{add}(0|y) & = y \\
\text{add}(n + 1|y) & = \text{succ}(\text{add}(n|y)) \\
\text{mul}(x, 0|) & = 0 \\
\text{mul}(x, n + 1|) & = \text{add}(x|\text{mul}(x, n)) \\
\text{sumlist}([],|) & = 0 \\
\text{sumlist}(w \cdot n|) & = \text{add}(n|\text{sumlist}(w|))
\end{align*}
\]
Real Pola!

data Nat → c = Zero: → c
| Succ: c → c;

add:: Nat | Nat → Nat
= n | m . fold h(x,y) as
{ Zero. y
| Succ x’. Succ (h(x’,y)) } in h(n,m);

sub:: Nat | Nat → Nat
= n | m . fold h(x,y) as
{ Zero. Zero
| Succ x’. case y of
| Succ y’. h(x’,y’) } in h(n,m);
Simple programs, cont’d

Consider the example of reversing a list:

\[
\begin{align*}
\text{rev}_1([[]]) &= [] \\
\text{rev}_1([w\cdot a [[]]) &= \text{rev}_1([w|\text{cons}([[]], a)])
\end{align*}
\]

We can define a higher-order version of this

\[
\text{rev}_2 : \text{list}_o(A_o) \rightarrow \text{list}_o(A) \rightarrow \text{list}_p(A)
\]

defined by:

\[
\begin{align*}
\text{rev}_2([[]]) &= \lambda x.x \\
\text{rev}_2([w\cdot a [[]]) &= \lambda x.\text{rev}_2(w)(\text{cons}([x, a]))
\end{align*}
\]

Then \(\text{rev}_1(w) = (\text{rev}_2(w))([]).\)
data List a → c = Nil: → c
| Cons: a,c → c ;

rev:: List a | → List a
= z . fold r(v,w) as { Nil.w
; Cons(a,v’) .r(v’,Cons(a,w)) }
in r(z,Nil);
data Tree a → c = Leaf: a → c
| Node: c,c → c ;
collect:: Tree a | → List a
    = t . fold c(v,w) as { Leaf.Cons(a,w) ; Node(t1,t2) .c(t1,c(t2,w)) } in c(t,Nil);
Not Pola!

```
exptree:: Nat | → Tree 1
  = n . fold big(x) as { Zero.Leaf ()
                     ; Succ(x') .Node(big(x'),big(x')) } in big(n);
```

(Collect leaves ...)
(Can build exptree with Leivant’s trees ... but can’t collect leaves!)
Why is the recursion rule so complicated?

- Must mimic safe recursion ...
- Datatypes built from modalities ...
- BUT datatypes are *not* modalities: in the p-world they are not even functors!
- This recursion allows “folding”, in a form similar to standard notation, which has some built-in higher-order aspects.
- Surmounts the objection of Colson without making the language explicitly higher-order: one can program functions such as subtraction efficiently and naturally!!
Kozen style recursion

One can translate the recursion box into a “Kozen” style recursion:

- Remove the p-context by making the function higher-order

$$\Gamma \mid \Delta, D(A) \leftrightarrow C \iff \Gamma \mid D(A) \vdash \Delta \circ C$$

(this step uses the strengths of the modalities)

- Replace the parametric arrow by the identity arrow.

Note the translation *requires* higher-order ...

$$\{ \Gamma \mid F_i(C, A) \vdash C \}_i$$

$$\Gamma, D(A) \vdash C$$

.... Kozen style recursion is subject to Colson’s objection!
Coinductive data

Pola

has coinductive data (with usual couniversal property) in the p-world ...

- ... this gives coinductive data which can be unfolded in (parametrically) constant time.
- Process people seem to be excited about this ...
Polytime soundness

The term logic admits an operational semantics in which programs are evaluated on inputs (maps from the final object in the opponent category) by value.

We show that given any opponent program:

\[ p : S \vdash_o t : T \]

there is a polynomial in the size of its inputs which bounds the running time. It is in this sense that our system is sound for polynomial time computations.

The fact that any polynomial time computable function is definable in our system can be proved by a directly programming a polynomially bounded Turing machine!
Why do it? Different (non-syntactic) perspective ...

What does it add? Source of P-time equality ...
and models.
Let $\mathbf{X}$ be a cartesian category then a $\mathbf{X}$-\textit{strong category} consists of a category $\mathbf{Y}$ and a \textit{module}:

$$\mathbf{X} \times \mathbf{Y} \hookrightarrow \mathbf{Y}$$

equipped with a \textit{strong composition}

$$f : (X_1, Y_1) \hookrightarrow Y_2 \quad g : (X_2, Y_2) \hookrightarrow Y_3$$

$$f; g : (X_1 \times X_2, Y_1) \hookrightarrow Y_3$$

and identity maps

$$i_Y : (1, Y) \hookrightarrow Y$$

which satisfy the natural equations.
An $\mathbb{X}$-strong functor $F : \mathbb{Y} \to \mathbb{Y}'$ between $\mathbb{X}$-strong categories consists of an ordinary functor $F : \mathbb{Y} \to \mathbb{Y}'$ and a morphism, also labeled $F$, on cross maps such that

$$f : (X, Y) \mapsto Y'$$

$$F(f) : (X, F(Y)) \mapsto F(Y')$$

preserving the strong composition, etc.

Transformations between strong functors are given by ordinary transformations between the ordinary functors $\alpha : F \to F'$ such that for cross maps $h$ we have

$$(1, \alpha)F'(h) = F(h)\alpha$$
An \( \mathbb{X} \)-strong category \( \mathbb{Y} \) has a tensor in case \( \mathbb{Y} \) has a tensor and the tensor works for cross maps as well:

\[
f : (X, Y_1) \leftrightarrow Y'_1 \quad g : (X, Y_2) \leftrightarrow Y'_2
\]

\[
f \otimes g : (X, Y_1 \otimes Y_2) \leftrightarrow Y'_1 \otimes Y'_2
\]

Again, there are various equations involving the structural maps which must be satisfied. The tensor unit \( 1 \in \mathbb{Y} \) is a final object if it is final in \( \mathbb{Y} \) and for any pair of objects \( (X, Y) \in \mathbb{X} \times \mathbb{Y} \), there is exactly one cross map

\[
(X, Y) \leftrightarrow 1
\]

which of course must factor as \((!, !)i_1\).
Affine strong categories

An $X$-strong category $Y$ with a tensor whose unit is final is said to be **affine**.

Affine $X$-strong categories provide the semantics of the basic logic of safety.
Bundle Fibrations

Given a $\mathbf{X}$-strong category $\mathbf{Y}$ one can form a “bundle” fibration

$$\partial : \tilde{\mathbf{Y}} \to \mathbf{X}; (X, Y) \mapsto X$$

where $\tilde{\mathbf{Y}}$ has:

- **Objects**: $(X, Y) \in \mathbf{X} \times \mathbf{Y}$ where $\partial(X, Y) = X$;
- **Maps**: $(x, h) : (X_1, Y_1) \to (X_2, Y_2)$ where $x : X_1 \to X_2$ in $\mathbf{X}$ and $h$ is a cross map $h : (X_1, Y_1) \smapsto Y_2$ and $\partial(f, h) = f$;
- **Identities**: $(1_X, (!, 1); i_Y) : (X, Y) \to (X, Y)$;
- **Composition**: $(x, h)(x', h') = (xx', (\Delta(1, x), 1)h; h')$.

It is straightforward to check that this is a category and furthermore a fibration (“simple” fibration).
Affine bundle Fibrations

This a 2-functor (reflexion):

- $X$-strong functor $\rightarrow$ morphism of fibration
- $X$-strong transformation $\rightarrow$ transformation of morphisms of fibrations

(One can almost recapture the original semantics from the fiber over 1 ...)

An affine bundle fibration is a fibration all of whose fibers are affine tensor categories and whose substitution functors preserve this structure (on the nose).

This gives an alternative way of viewing the semantics of the basic logic of safety as a fibration.
An $X$-strong category $Y$ has a *lift* if for each $Y \in Y$ there is an object $\uparrow(Y) \in X$ and a cross map

$$\epsilon_Y : (\uparrow(Y), 1) \rightarrow Y$$

satisfying the following universal property: whenever $h : (X, 1) \rightarrow Y$ there is a unique map $h^b : X \rightarrow \uparrow(Y)$ making

$$\begin{array}{ccc}
(X, 1) & \xrightarrow{h} & Y \\
\downarrow & & \downarrow \\
(\uparrow(Y), 1) & \xrightarrow{\epsilon_Y} & (h^b, 1)
\end{array}$$

commute.
Lifts are comprehensions

For the bundle fibration this simply means means one has comprehension ...

\[ \delta \vdash 1 \vdash (\_ \_ \_ ) \]
Polarized operators

A polarized operator on an \( X \)-strong category \( Y \) consists of a pair of \( X \)-strong functors \( F_o : X^n \to X \) and \( F_p : Y^n \to Y \), and a map of the cross maps \( F_{op} \):

\[
\begin{align*}
F_o(x_1, \ldots, x_n) : F_o(X_1, \ldots, X_n) &\to F_o(X'_1, \ldots, X'_n) \\
F_{op}(h_1, \ldots, h_n) : (F_o(X_1, \ldots, X_n), \vec{Y}) &\leftrightarrow F_p(Y'_1, \ldots, Y'_n) \\
F_p(y_1, \ldots, y_n) : F_p(Y_1, \ldots, Y_n) &\to F_p(Y'_1, \ldots, Y'_n)
\end{align*}
\]

which preserve the compositions. When there is a lift, a polarized operator must preserve the lift structure.
Examples (polarized functors = modalities):

- The constants $K^Y$: let $Y \in \mathcal{Y}$ and define $K^Y$ to have
  $K_p^Y(Y') = Y$, $K_o^Y(X) = \uparrow(Y)$ and $K_{op}^Y(h) = (1, !)\epsilon_Y$;

$$h : (X, Y') \leftrightarrow Y'' \quad \frac{K_o^Y(h) = (1, !)\epsilon_Y : (\uparrow(Y), Y') \leftrightarrow Y}{K_{op}}$$

- On any $\mathcal{X}$-strong category with products and a lift, the product functor gives a polarized operator;

- The coproduct is a polarized operator in an $\mathcal{X}$-strong category $\mathcal{Y}$ with a lift that has coproducts.

- Polarized operators compose so that from these base operators one can derive new operators.
Let $F_p(C, A) : \mathcal{Y}^{m+1} \rightarrow \mathcal{Y}$ be a polarized operator on an $\mathcal{X}$-strong category $\mathcal{Y}$. Then a polarized inductive datatype, $D_p(A)$, is a fixed point of this functor in $\mathcal{Y}$. So there is an isomorphism

$$c : F_p(D_p(A), A) \rightarrow D_p(A)$$

Note that this implies that $\uparrow(D_p(A))$ is a fixed point of $F_o$ as

$$\begin{align*}
(F_o(\uparrow(D_p(A)), \uparrow(A)), 1) &\leftrightarrow F_p(D_p(A), A) \\
(F_o(\uparrow(D_p(A)), \uparrow(A)), 1) &\rightarrow D_p(A) \\
F_o(\uparrow(D_p(A)), \uparrow(A)) &\rightarrow \uparrow(D_p(A))
\end{align*}$$
Polarized inductive data

In addition to being a fixed point the following universal property must hold:

\[
(\mathbb{X} \times \uparrow (F_i(D(A), A)), 1) \xrightarrow{(1 \times \uparrow(C_i), 1)} (\mathbb{X} \times \uparrow(D(A)), 1)\]

\[
(1, F_i(\text{fold}(f))) \xrightarrow{\text{fold}(f)} (\mathbb{X} \times \uparrow(F_i(D(A), A)), F_i(C, A)) \xrightarrow{f} C
\]

we may call this is **comprehended initial**.

We need to assume the fibers are closed in order to get the full recursion specified by the “boxes” in the logic.
A categorical semantics for P-time is given by the initial category, $\mathbb{P}$, such that it is:

1. A cartesian (in fact, distributive) category $\mathbb{P}$ with a closed affine bundle fibration

$$\vartheta : \widehat{\mathbb{P}} \to \mathbb{P}$$

which has coproducts and comprehension.
2. With fixed points for polynomial functors which are comprehended initial.
BTW: If there is anywhere where complexity classes should clearly separate it is in these models ...