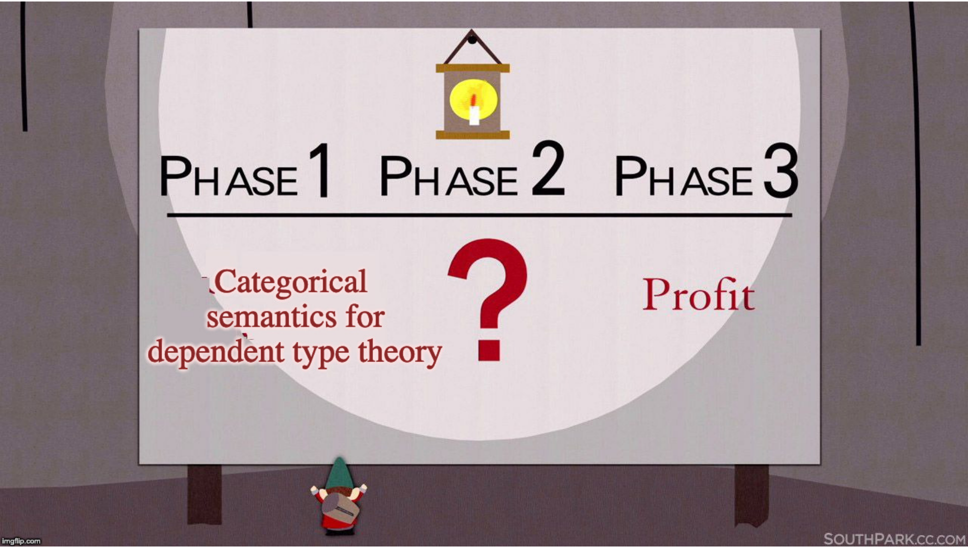
Dependent Types Made Difficult

Mark Hopkins

@antiselfdual

mjhopkins.github.io

What is the categorical semantics of dependent type theory?

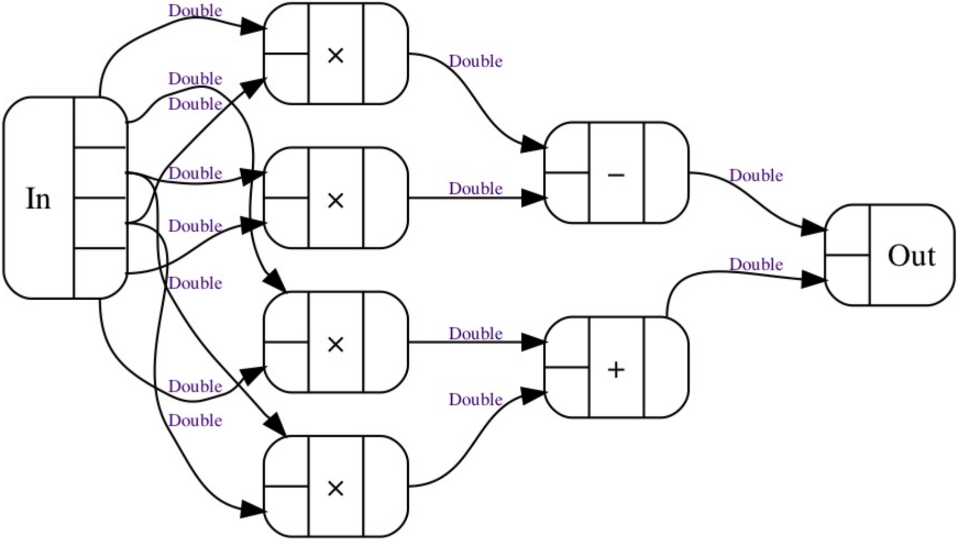

PHASE 1
PHASE 2
PHASE 3
Categorical semantics for dependent type theory
?
Profit
imgflip.com
SOUTHPARK.CC.COM

Compiling to Categories (Elliott, 2017)

Take in Haskell source and spit out ...

Compiling to Categories (Elliott, 2017)

- computation graphs
  - diagrams
  - circuit descriptions (VHDL, Verilog)
- linear maps
- automatic differentiation
- incremental computation
- interval analysis
- Kleisli category for any monad e.g. probabilistic programming
- graphics (GLSL)
- syntax
- products of the above
- tons more . . . see https://github.com/conal/concat


In
×
×
×
×
−
+
Out
Double
Double
Double
Double
Double
Double
Double
Double
Double
Double
Double
Double
Double
Double

How does this work??

Categorical semantics

The simply-typed lambda calculus is the *internal language* of *Cartesian closed categories*.

which means

We can interpret the lambda calculus in any CCC.

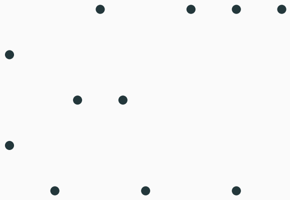
Categorical semantics

Dependent type theory is the *internal language* of ???.

which means

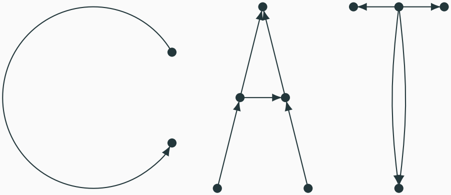
We can interpret dependently typed programs in any ???.

Categories



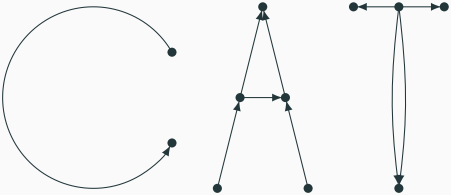
$\mathrm{Ob}\, C : \mathbf{Set}$

Categories



$\mathrm{Ob}\,\mathrm{C} : \mathbf{Set} \qquad C(a, b) = \mathrm{Hom}(a, b) : \mathbf{Set}$

Categories



$\operatorname{Ob} \mathrm{C} : \mathbf{Set} \qquad C(a,b) = \operatorname{Hom}(a,b) : \mathbf{Set}$

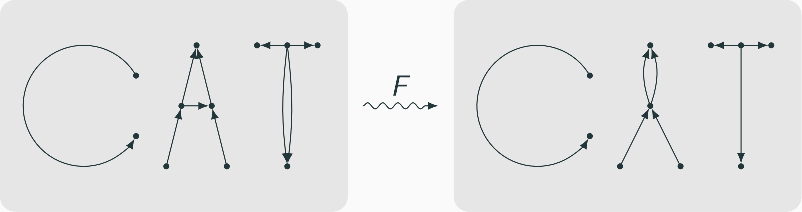
$\circ : C(b,c) \times C(a,b) \to C(a,c) \quad 1_a \in C(a,a)$

$1_b \circ f = f = f \circ 1_a \quad h \circ (g \circ f) = (h \circ g) \circ f$

Functors

Functors

A functor from $C \to D$ draws a picture of C in D.



$$F : \mathrm{Ob}\, C \to \mathrm{Ob}\, D$$

$$F : C(a, b) \to D(Fa, Fb) \text{ i.e. fmap}$$

$$F(1) = 1$$

$$F(f \circ g) = F(f) \circ F(g)$$

Cartesian closed categories

A Cartesian closed category is a category with function objects.

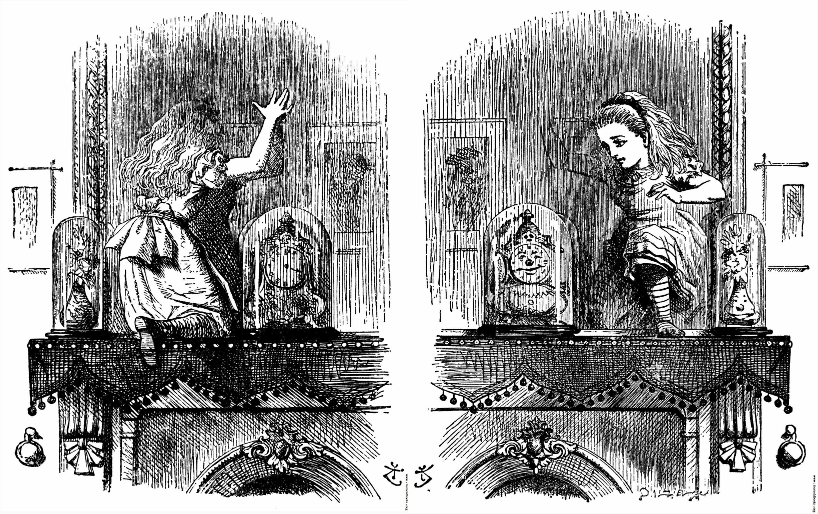
$$\mathcal{C}(A \times B, C) \cong \mathcal{C}(A, C^B)$$

In Haskell, this isomorphism is called "curry".

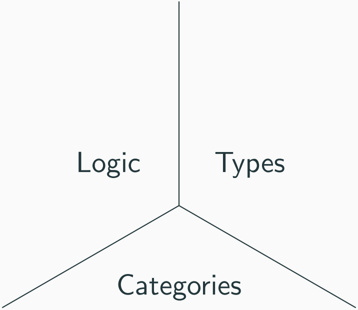
```
curry :: ((a,b) -> c) -> a -> b -> c
curry f a b = f (a, b)

uncurry :: (a -> b -> c) -> (a, b) -> c
uncurry f (a, b) = f a b
```

Categorical semantics

Curry-Howard correspondence



Curry-Howard-Lambek correspondence



Categorical semantics

$$\begin{tikzcd}
\text{Type theories} \arrow[rr, bend left, "\text{Syntactic category}"] \arrow[rr, phantom, "\perp"] & & \text{Categories} \arrow[ll, bend left, "\text{Internal language}"]
\end{tikzcd}$$

Advantages

- The internal language is valid in every model
- More easily prove properties about a type theory using CT
- CT proofs using the internal language can be easier
- The CT can illuminate the TT, and vice versa
- Can use the internal language to define internal structures
- Refinement of denotational semantics

Adjunctions

Adjunctions

$$F \dashv U : \mathcal{C} \to \mathcal{D}$$

$$\begin{tikzcd} \mathcal{C} \arrow[r, bend left, "F"] \arrow[r, phantom, "\perp"] & \mathcal{D} \arrow[l, bend left, "U"] \end{tikzcd}$$

$$\mathcal{D}(Fc, d) \cong \mathcal{C}(c, Ud)$$

$$\frac{Fc \to d \text{ in } \mathcal{D}}{c \to Ud \text{ in } \mathcal{C}}$$

Example: free monoid

The free monoid on an alphabet Λ is the set Λ^* of words in Λ.

$$\frac{\Lambda^* \to m \text{ in } \mathbf{Mon}}{\Lambda \to m \text{ in } \mathbf{Set}}$$

Why?

A monoid homomorphism has to respect multiplication, so

$$f(abcd\cdots) = f(a)f(b)f(c)\cdots$$

Example: currying

The functor $- \times b$ is left adjoint to $(-)^b$.

$$\frac{a \times b \to c \text{ in } \mathbf{Type}}{a \to c^b \text{ in } \mathbf{Type}}$$

```
curry :: ((a,b) -> c) -> a -> b -> c
curry f a b = f (a, b)

uncurry :: (a -> b -> c) -> (a, b) -> c
uncurry f (a, b) = f a b
```

Example: quantifiers $\exists_y \dashv w_y \dashv \forall_y$

$$\begin{aligned} w_y : \mathrm{Form}(\bar{x}) &\to \mathrm{Form}(\bar{x}, y) \\ \phi(\bar{x}) &\mapsto \phi(\bar{x}) \end{aligned}$$

$$\frac{\bar{x}, y \vdash w_y\phi(\bar{x}) \Rightarrow \psi(\bar{x}, y)}{\bar{x} \vdash \phi(\bar{x}) \Rightarrow \forall_y \psi(\bar{x}, y)}$$

$$\frac{\bar{x} \vdash \exists_y \psi(\bar{x}, y) \Rightarrow \phi(\bar{x})}{\bar{x}, y \vdash \psi(\bar{x}, y) \Rightarrow w_y\phi(\bar{x})}$$

Compiling to CCCs

- Categorical semantics for STLC in CCCs
- Lots of CCCs
- GHC Core's System FC can (mostly) be converted to STLC
- GHC plugin, rewrite rules
- Convert Haskell src to VHDL, diagrams, etc

There's more to type theory than STLC!

- polymorphic types
- existential types
- universal types
- type classes
- union and intersection types
- quotient types
- dependent types
- refinement types
- homotopy type theory
- ...

Dependent types

Dependent types

A dependent type is one that contains free variables.

$$\tau : \mathrm{Type} \quad x : \tau \vdash P(x) : \mathrm{Type}$$

It *depends on* terms.

Dependent types

$$n : \mathrm{Nat} \vdash \mathrm{IsEven}\, n : \mathrm{Type}$$

```
data IsEven (n : Nat) : Type where
  ZEven : IsEven 0
  SSEven : IsEven n -> IsEven (n + 2)
```

Dependent types

$$n : \mathrm{Nat}, a : \mathrm{Type} \vdash \mathrm{Vect}_n(a) : \mathrm{Type}$$

```
data Vect (n : Nat) (a : Type) : Type where
  Nil : Vect 0 a
  (::) : a -> Vect n a -> Vect (n + 1) a
```

Dependent product type

```
replicate : (n : Nat) -> a -> Vect n a
```

$$\begin{aligned}\text{replicate} : &\prod_{a:\text{Type}} \prod_{n:\mathbb{N}} \prod_{x:a} \text{Vect}_n(a) \\ &= \prod_{a:\text{Type}} \prod_{n:\mathbb{N}} a \to \text{Vect}_n(a)\end{aligned}$$

$$a \to b :\equiv \prod_{\_:a} b$$

Dependent sum type

```
evenLenLists = (l : List Int ** m : Nat ** length l = 2 * m)
```

$$\text{evenLenLists} = \sum_{l:\text{List}(\text{Int})} \sum_{m:\mathbb{N}} \text{length}(l) = 2m$$

$$(a, b) :\equiv \sum_{\_:a} b$$

Dependent types are the internal language of *locally Cartesian closed categories*.

Semantics

Semantics

Objects: closed types

Arrows: terms in context

also arrows: functions

$$f : a \to b$$

$$[\![f]\!] : [\![a]\!] \to [\![b]\!]$$

Semantics for dependent types

$$
\begin{aligned}
&\mathsf{IsEven} : \mathbb{N} \to \textbf{Set} \\
&\mathsf{IsEven}\,0 = \{\text{zeroEven}\} \\
&\mathsf{IsEven}\,1 = \{\} \\
&\mathsf{IsEven}\,2 = \{\text{twoEven}\} \\
&\mathsf{IsEven}\,3 = \{\} \\
&\mathsf{IsEven}\,4 = \{\text{fourEven}\} \\
&\qquad\vdots
\end{aligned}
$$

Semantics for dependent types

$$\mathsf{CustomerOf} : \mathsf{Bank} \to \textbf{Set}$$

$$\mathsf{CustomerOf}\ \mathsf{ANZ} = \{\text{Alice, Albert, Anne-Marie}\}$$

$$\mathsf{CustomerOf}\ \mathsf{CBA} = \{\text{Calvin, Colin}\}$$

$$\mathsf{CustomerOf}\ \mathsf{NAB} = \{\text{Nathan}\}$$

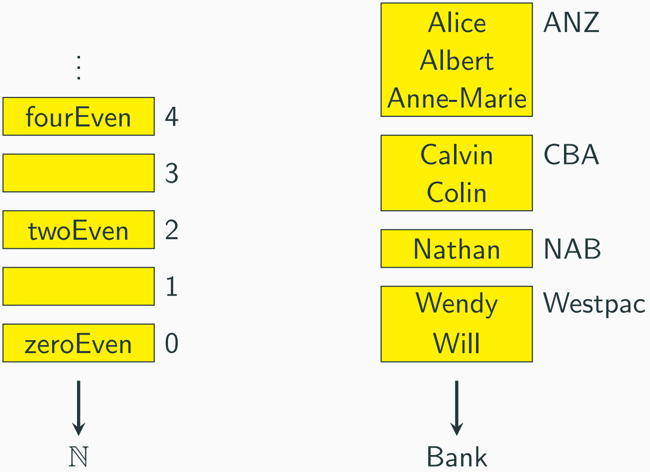
$$\mathsf{CustomerOf}\ \mathsf{Westpac} = \{\text{Wendy, Will}\}$$

Thinking backwards

$$p : \mathsf{IsEven}\ \_ \to \mathbb{N}$$
$$p\ \mathit{zeroEven} = 0$$
$$p\ \mathit{twoEven} = 2$$
$$p\ \mathit{fourEven} = 4$$
$$\vdots$$

$$p : \mathsf{CustomerOf}\ \_ \to \mathsf{Bank}$$
$$p\,\mathrm{Alice} = \mathsf{ANZ}$$
$$p\,\mathrm{Albert} = \mathsf{ANZ}$$
$$p\,\mathrm{Anne\text{-}Marie} = \mathsf{ANZ}$$
$$p\,\mathrm{Calvin} = \mathsf{CBA}$$
$$p\,\mathrm{Colin} = \mathsf{CBA}$$
$$p\,\mathrm{Nathan} = \mathsf{NAB}$$
$$p\,\mathrm{Wendy} = \mathsf{Westpac}$$
$$p\,\mathrm{Will} = \mathsf{Westpac}$$

Semantics for dependent types



$\mathbf{Set}/\mathbb{N}$ $\mathbf{Set}/\mathrm{Bank}$

Over-categories

$$\mathcal{C}/X \text{ where } X \in \mathcal{C}$$

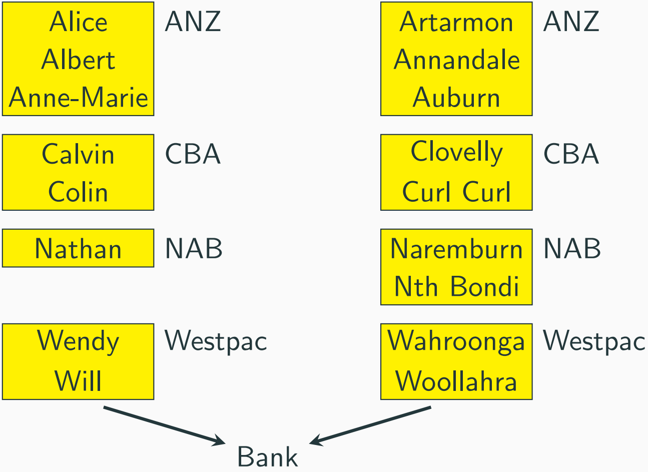
Objects:
$$\begin{tikzcd} A \arrow[d, "p"] \\ X \end{tikzcd}$$

fibre $(A)_x = p^{-1}x$ in **Set**

Arrows:
$$\begin{tikzcd} A \arrow[rr, "f"] \arrow[dr, "p"'] & & B \arrow[dl, "q"] \\ & X & \end{tikzcd}$$

section $s : X \to A$
such that $p \circ s = \mathrm{id}$

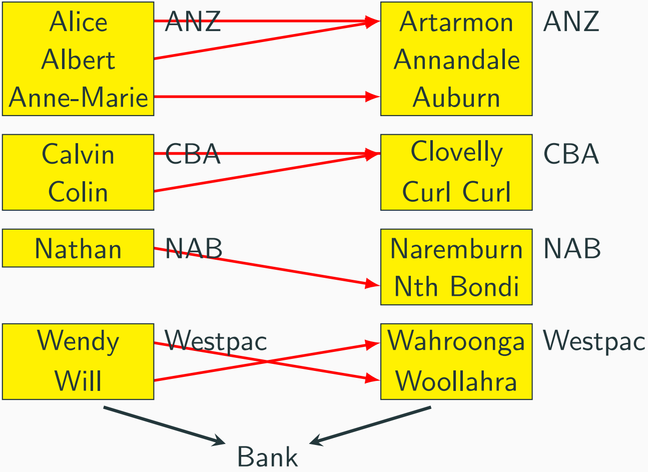
Over-categories



fibre $(A)_x = p^{-1}x$

section $s : X \to A$ such that $p \circ s = \mathrm{id}$

Over-categories



fibre $(A)_x = p^{-1}x$

section $s : X \to A$ such that $p \circ s = \mathrm{id}$

Substitution

$$\mathsf{bankOf} : \mathsf{Branch} \to \mathsf{Bank}$$

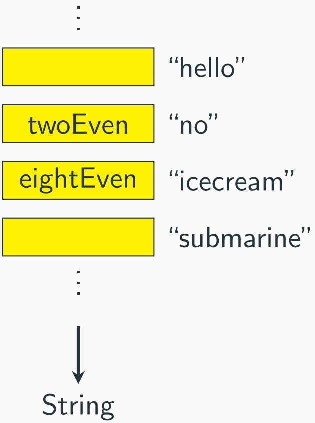
$$b : \mathsf{Bank} \vdash \mathsf{CustomerOf}\ b : \mathsf{Type}$$

$$br : \mathsf{Branch} \vdash \mathsf{CustomerOf}(\mathsf{bankOf}\ br) : \mathsf{Type}$$

Alice Albert	Artarmon
	Annandale
Anne-Marie	Auburn
Calvin Colin	Clovelly
	Curl Curl
	Naremburn
Nathan	Nth Bondi
Will	Wahroonga
Wendy	Woollahra

↓

Branch



$$s : \text{String} \vdash \text{IsEven}\ (\text{length}\ s) : \text{Type}$$

Substitution = pullback aka change of base = join

$$\mathrm{bankOf} : \mathrm{Branch} \to \mathrm{Bank}$$

$$\mathrm{bankOf}^* : \mathbf{Set} \,/\, \mathrm{Bank} \to \mathbf{Set} \,/\, \mathrm{Branch}$$

Substitution = pullback aka change of base = join

$$
\begin{tikzcd}
T \arrow[drr, bend left, "f"] \arrow[dr, dashed, "\exists!"] \arrow[ddr, bend right, "g"'] & & \\
& \text{Branch} \times_{\text{Bank}} \text{Customer} \arrow[r] \arrow[d, "p^*"] & \text{Customer} \arrow[d, "p"] \\
& \text{Branch} \arrow[r, "\text{bankOf}"] & \text{Bank}
\end{tikzcd}
$$

$$
\begin{aligned}
\text{Branch} \times_{\text{Bank}} \text{Customer} &= \text{bankOf}^* \, \text{Customer} \\
&= \{(br, c) | \text{bankOf}(br) = p(c)\} \\
&= \text{select * from Branch inner join Customer} \\
&\quad\ \text{on Branch.bank} = \text{Customer.bank}
\end{aligned}
$$

Dependent sum

$$f : X \to Y$$

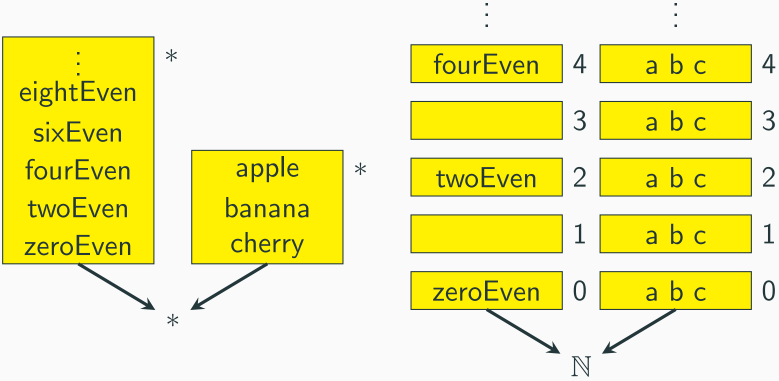
$$\sum_f : \mathbf{Set}/X \to \mathbf{Set}/Y$$

$$\begin{tikzcd} A \arrow[d, "p"] \\ X \arrow[d, "f"] \\ Y \end{tikzcd}$$

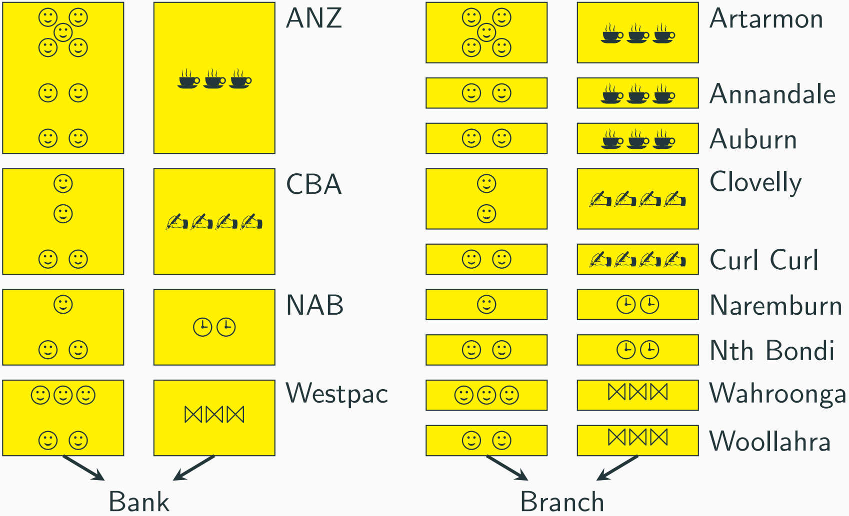
$$\sum_f \dashv f^*$$

$\sum_{x:X} := \sum_{!_X} \quad !_X : X \to *$

$$\mathbf{Set}/*\left(\sum_f p, q\right) \cong \mathbf{Set}/X(p, f^* q)$$



$\sum_{\mathrm{bankOf}}$: **Set** / Branch $\to$ **Set** / Bank



Dependent product

A category is **locally Cartesian closed**
if f^* also has a right adjoint
for all arrows f.

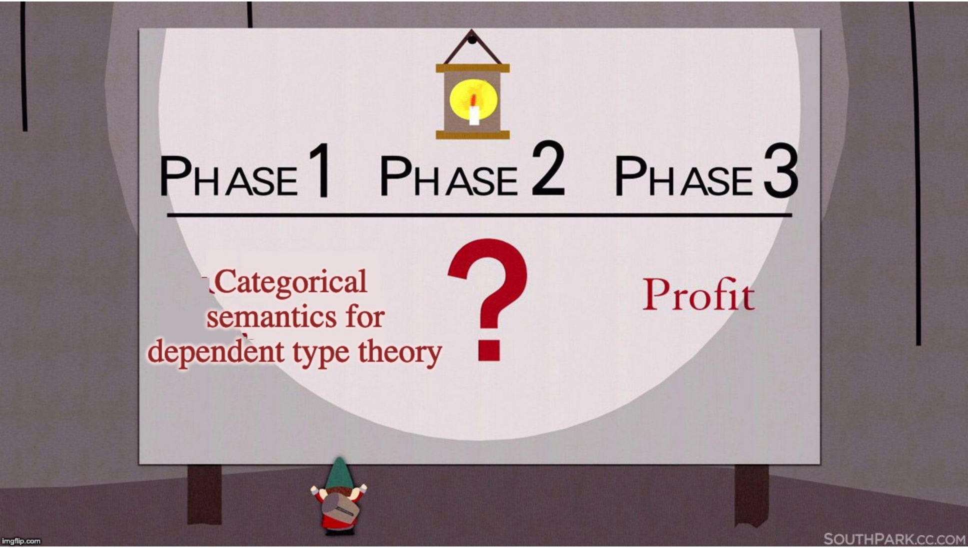
Lemma
In this case all the slice categories $\mathcal{C}/X$ are Cartesian closed.

Categorical semantics

Dependent type theory is the *internal language* of *locally Cartesian closed categories*.

which means

We can interpret dependently typed programs in any LCCC.


PHASE 1
PHASE 2
PHASE 3
Categorical semantics for dependent type theory
?
Profit
imgflip.com
SOUTHPARK.CC.COM

Compiling to LCCCs

- Categorical semantics for DTs in LCCCs
- Lots of LCCCs
- Something something Agda, Idris, Lean, ...
- **Profit**

LCCCs

Set is locally Cartesian closed.

Dependent product

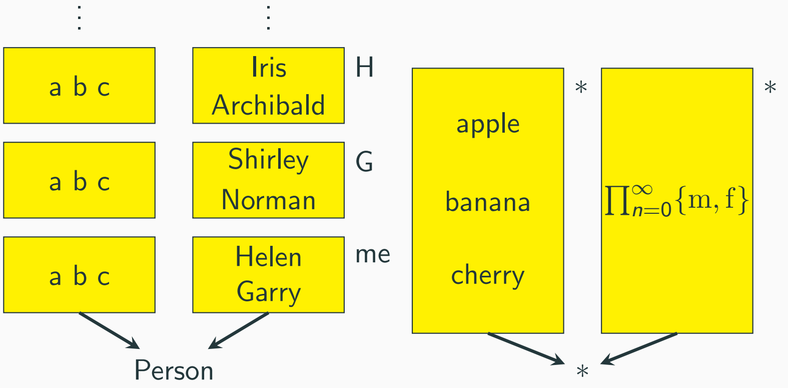
$$f : X \to Y$$

$$\prod_f : \mathbf{Set}\,/X \to \mathbf{Set}\,/Y$$

$$f^* \dashv \prod_f$$

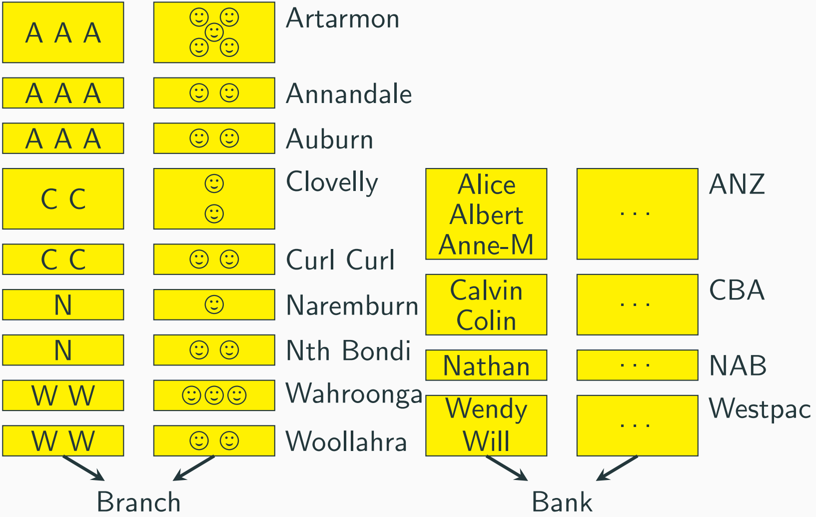
$\prod_{x:X} := \prod_{!_X} \quad !_X : X \to *$

$$\mathbf{Set}\,/X(f^*p, q) \cong \mathbf{Set}\,/*\,(p, \textstyle\prod_f q)$$



$$\dfrac{p : Person \vdash Fruit \to ParentOf(p)}{\vdash Fruit \to \prod_{p:Person} ParentOf(p)}$$

$\prod_{\text{bankOf}} : \textbf{Set} / \text{Branch} \to \textbf{Set} / \text{Bank}$



$$(\textstyle\prod_{\text{bankOf}} \text{Staff})_{ANZ} = \{s : (\text{Branch})_{\text{ANZ}} \to \text{Staff} \mid \forall br.\, \text{branchOf}(s(br)) = br\}$$

Subtleties

$$c : \mathsf{Customer} \vdash \mathsf{IsEven}(\mathsf{length}(\mathsf{firstName}(c)))$$

- substitute $s := \mathsf{firstName}(c)$ into $\mathsf{IsEven}(\mathsf{length}(s))$
- substitute $n := \mathsf{length}(\mathsf{firstName}(c))$ into $\mathsf{IsEven}\ n$

Subtleties

- display map categories
- contextual categories
- categories with families
- categories with attributes
- Awodey's "natural model"

References

R. A. G. Seely, *Locally cartesian closed categories and type theory*

Martin Hofman, *Syntax and semantics of dependent types*

Alexandre Buisse, *Categorical models of dependent type theory*

Awodey, *Category Theory*, 7.29 and 9.7

MacLane and Moerdijk, *Sheaves in Geometry and Logic*, IV.7

Bart Jacobs, *Categorical Logic and Type Theory*

The nLab, ncatlab.org/nlab

Compiling to Categories (YouTube)