Algebraic data (types and semirings)

February 2018

Last time: programming with equalities

$$a \equiv b$$

This time: arithmetic with types

$$A \times (B + C) = D$$

Recap: various views of types

Types **classify terms** (and every term has a "best type")

$$\Gamma \vdash M : A$$

Types **induce relations** (and polymorphic functions preserve relations)

$$\begin{aligned}
&\forall B_1. \ldots \forall B_n. \forall x : (\forall \alpha. T[\alpha, B_1, \ldots, B_n]). \\
&\quad \forall \gamma. \forall \delta. \forall \rho \subset \gamma \times \delta. \\
&\qquad [\![T]\!][\rho, =_{B_1}, \ldots, =_{B_n}](x[\gamma],\ x[\delta])
\end{aligned}$$

Types **correspond to propositions** (for which terms are evidence)

$$\Gamma \vdash A$$

The three-part Curry-Howard correspondence

Types correspond to **propositions**

Programs correspond to **proofs**

Evaluation corresponds to **proof simplification**

Why might we consider other views besides Curry-Howard?

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \wedge A}\ \wedge\text{-intro} \qquad \frac{\Gamma \vdash A \wedge A}{\Gamma \vdash A}\ \wedge\text{-elim-1}$$

Propositions A and $A \wedge A$ are **interderivable** . . .
. . . but **types** A and $A \times A$ are **not equivalent**.

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee A}\ \vee\text{-intro-1} \qquad \frac{\Gamma \vdash A \vee A}{\Gamma \vdash A}\ \vee\text{-elim}$$

Propositions A and $A \vee A$ are **interderivable** . . .
. . . but **types** A and $A + A$ are **not equivalent**.

Types and cardinality, continued

So if $A \equiv B$, i.e.

$$\Gamma, A \vdash B \qquad \Gamma, B \vdash A$$

then we can build terms that **convert** between A and B

$$\Gamma, x{:}A \vdash M : B \qquad \Gamma, y{:}B \vdash N : B$$

but A and B may not be **equivalent** in the programming language.

In **logic** we ask: *is A inhabited*?

In **programming** we ask: *how many values have type A*?
(and much more besides!)

From **inhabitance**
to **inhabitants**

A suggestive notation

$$0 \qquad 1 \qquad A \times B \qquad A + B \qquad \Pi x : P.A$$

A suggestive notation

$$0 \qquad 1 \qquad A \times B \qquad A + B \qquad \Pi_{x \in P} A(x)$$

A suggestive notation

$$0 \qquad 1 \qquad A \times B \qquad A + B \qquad \Pi_{x \in P} A(x)$$

$$0 + 1$$

$$(1 + 1) + 0 \times 1$$

$$\Pi_{x \in (1+1)} (y + 1)$$

Algebraic data types in OCaml

```
type 'a list =
   Nil : 'a list
 | Cons : 'a * 'a list -> 'a list
```

$$\lambda\alpha.\mu\ell.1 + (\alpha \times \ell)$$

Algebraic data types in OCaml

type parameter

```
type 'a list =
   Nil : 'a list
 | Cons : 'a * 'a list -> 'a list
```

$$\lambda\alpha.\mu\ell.1 + (\alpha \times \ell)$$

introduce recursive type

type parameter

```
type 'a list =
   Nil : 'a list
 | Cons : 'a * 'a list -> 'a list
```

$$\lambda\alpha.\mu\ell.1 + (\alpha \times \ell)$$

Algebraic data types in OCaml

introduce recursive type

type parameter

```
type 'a list =
   Nil : 'a list
 | Cons : 'a * 'a list -> 'a list
```

0-argument constructor

$$\lambda\alpha.\mu\ell.1 + (\alpha \times \ell)$$

Algebraic data types in OCaml

introduce recursive type

type parameter

```
type 'a list =
   Nil : 'a list
 | Cons : 'a * 'a list -> 'a list
```

binary sum

$$\lambda\alpha.\mu\ell.1 + (\alpha \times \ell)$$

Algebraic data types in OCaml

introduce recursive type

type parameter

```
type 'a list =
   Nil : 'a list
 | Cons : 'a * 'a list -> 'a list
```

2-argument constructor

$$\lambda\alpha.\mu\ell.1 + (\alpha \times \ell)$$

Algebraic data types in OCaml

introduce recursive type

type parameter

```
type 'a list =
    Nil : 'a list
  | Cons : 'a * 'a list -> 'a list
```

recursive occurrence

$$\lambda\alpha.\mu\ell.1 + (\alpha \times \ell)$$

Fixed point **equation**:

$$\mu\ell.A \quad = \quad A[\ell := \mu\ell.A]$$

Unrolling the fixed point for lists of α:

$$\begin{aligned}\mu\ell.1 + (\alpha \times \ell) &= 1 + (\alpha \times (\mu\ell.1 + (\alpha \times \ell))) \\ &= 1 + (\alpha \times (1 + (\alpha \times (\mu\ell.1 + (\alpha \times \ell))))) \\ &= \ldots\end{aligned}$$

Lists correspond to power series:

```
'a list ≃ unit
        + 'a                (* × unit *)
        + ('a * 'a)         (* × unit *)
        + ('a * 'a * 'a)    (* × unit *)
        + ...
```

$$0 \qquad 1 \qquad a + b \qquad a \times b$$

Algebraic laws

$$(a + b) + c = a + (b + c) \qquad \text{(+-assoc)}$$

$$0 + a = a + 0 = a \qquad \text{(+-id)}$$

$$a + b = b + a \qquad \text{(+-commut)}$$

$$(a \times b) \times c = a \times (b \times c) \qquad \text{(×-assoc)}$$

$$1 \times a = a \times 1 = a \qquad \text{(×-id)}$$

$$a \times b = b \times a \qquad \text{(×-commut)}$$

$$0 \times a = a \times 0 = 0 \qquad \text{(×-annihil)}$$

$$a \times (b + c) = (a \times b) + (a \times c) \qquad \text{(distrib)}$$

First, approximate types as **sets**.

The set bool has two elements:

```
bool = { false , true }
```

The set `pbool` = $\forall\alpha.\alpha \to \alpha \to \alpha$ has two elements:

$$\texttt{pbool} = \{\ \Lambda\alpha.\lambda\texttt{x}{:}\alpha.\lambda\texttt{y}{:}\alpha.\texttt{x},\ \Lambda\alpha.\lambda\texttt{x}{:}\alpha.\lambda\texttt{y}{:}\alpha.\texttt{y}\ \}$$

Simplifying further, **types are cardinalities**

```
bool = 2
pbool = 2
```

and type equivalence means set isomorphism (i.e. bijection).

Cardinality of types

Semiring elements in OCaml and their cardinalities:

```
type zero = {z: 'a.'a}  (* |zero| = 0 *)

type unit = Unit : unit (* |unit| = 1 *)

type bool =             (* |bool| = 2 *)
  False : bool
| True : bool

type ('a,'b) sum =      (* |(a,b) sum| = |a| + |b| *)
  Inl : 'a -> ('a,'b) sum
| Inr : 'b -> ('a,'b) sum

type ('a,'b) pair =     (* |(a,b) pair| = |a| × |b| *)
  Pair : 'a * 'b -> ('a,'b) pair
```

Set isomorphisms

$$a \times 1 = a \qquad (\times\text{-id})$$

For each equation a pair of functions converts between the sides:

```
let times_id1 : 'a.('a * unit) -> 'a =
 fun (x, ()) -> x

let times_id2 : 'a.'a -> ('a * unit) =
 fun x -> (x, ())
```

Function pairs form **isomorphisms**: $f\ (g\ x) = x$ and $g\ (f\ y) = y$:

```
times_id1 (times_id2 x)  =  x
times_id2 (times_id1 x)  =  x
```

Isomorphisms and structure

We're ignoring interesting structure, e.g. **multiple isomorphisms**:

```
val not : bool -> bool
not (not x) = x
```

```
val id_bool : bool -> bool
id_bool (id_bool x) = x
```

Q: are these sometimes interchangeable?

And a given isomorphism may have **several proofs**:

$$
\begin{aligned}
(a+0)+b &= (\text{+-commut}) \\
&\quad (0+a)+b \\
&= (\text{+-id}) \\
&\quad a+b
\end{aligned}
\qquad
\begin{aligned}
(a+0)+b &= (\text{+-assoc}) \\
&\quad a+(0+b) \\
&= (\text{+-id}) \\
&\quad a+b
\end{aligned}
$$

Q: what does it mean for proofs to be equivalent?[1]

[1] See *Computing with Semirings and Weak Rig Groupoids (Carette & Sabry, 2016)*

`a` $\rightarrow$ `b` ?

Functions and cardinality

How many (pure) inhabitants does $a \rightarrow b$ have? Let's count!

$|$`unit` $\rightarrow$ `unit`$|$ `= 1`

```
fun Unit -> Unit
```

$|$`unit` $\rightarrow$ `bool`$|$ `= 2`

```
fun Unit -> False
fun Unit -> True
```

$|$`bool` $\rightarrow$ `unit`$|$ `= 1`

```
fun _ -> Unit
```

$|$`bool option` $\rightarrow$ `bool`$|$ `= 8`

```
function Some False -> False
       | Some True  -> False
       | None       -> False
function Some False -> False
       | Some True  -> False
       | None       -> True
...
```

In general ...

How many (pure) inhabitants does a $\to$ b have? Let's count!

|`unit` $\to$ `unit`| = 1 $= 1^1$

```
fun Unit -> Unit
```

|`unit` $\to$ `bool`| = 2 $= 2^1$

```
fun Unit -> False
fun Unit -> True
```

|`bool` $\to$ `unit`| = 1 $= 1^2$

```
fun _ -> Unit
```

|`bool option` $\to$ `bool`| = 8 $= 2^{(2+1)}$

```
function Some False -> False
       | Some True  -> False
       | None       -> False
function Some False -> False
       | Some True  -> False
       | None       -> True
...
```

In general ...

Functions and cardinality

How many (pure) inhabitants does $a \to b$ have? Let's count!

$|\texttt{unit} \to \texttt{unit}| = 1 \quad = 1^1$

```
fun Unit -> Unit
```

$|\texttt{unit} \to \texttt{bool}| = 2 \quad = 2^1$

```
fun Unit -> False
fun Unit -> True
```

$|\texttt{bool} \to \texttt{unit}| = 1 \quad = 1^2$

```
fun _ -> Unit
```

$|\texttt{bool option} \to \texttt{bool}| = 8 \quad = 2^{(2+1)}$

```
function Some False -> False
       | Some True  -> False
       | None       -> False
function Some False -> False
       | Some True  -> False
       | None       -> True
...
```

In general ...

Functions and cardinality

How many (pure) inhabitants does a $\rightarrow$ b have? Let's count!

$|\texttt{unit} \rightarrow \texttt{unit}| = 1 \quad = 1^1$

```
fun Unit -> Unit
```

$|\texttt{unit} \rightarrow \texttt{bool}| = 2 \quad = 2^1$

```
fun Unit -> False
fun Unit -> True
```

$|\texttt{bool} \rightarrow \texttt{unit}| = 1 \quad = 1^2$

```
fun _ -> Unit
```

$|\texttt{bool option} \rightarrow \texttt{bool}| = 8 \quad = 2^{(2+1)}$

```
function Some False -> False
       | Some True  -> False
       | None       -> False
function Some False -> False
       | Some True  -> False
       | None       -> True
...
```

In general a$\rightarrow$ b has $|b|^{|a|}$ inhabitants

Equations for exponents

$$
\begin{aligned}
a^1 &= a \\
a^0 &= 1 && (a \neq 0) \\
a^{b+c} &= a^b \times a^c \\
(a^c)^b &= a^{b \times c} \\
(b \times c)^a &= b^a \times c^a
\end{aligned}
$$

In OCaml notation:

`unit -> a`	$\simeq$	`a`	
`zero -> a`	$\simeq$	`unit`	(a $\not\simeq$ zero)
`(b,c) sum -> a`	$\simeq$	`(b -> a) * (c -> a)`	
`b -> (c -> a)`	$\simeq$	`(b * c) -> a`	
`a -> (b * c)`	$\simeq$	`(a -> b) * (a -> c)`	

Π types?

$\Pi x : P.A$?

Example (exponents): $A \to B$ abbreviates $\Pi x : A.B$. So

$$|\Pi x : A.B| \quad = \quad |A \to B| \quad = \quad |B|^{|A|} \qquad (x \notin \mathsf{fv}(B))$$

Example (A is `Bool`):

$$|\Pi x : \mathsf{Bool}.\texttt{if x then Bool else Unit}| \quad = \quad |\texttt{Bool}| \times |\texttt{Unit}| \quad = \quad 2$$

The two inhabitants:

```
λb:Bool.if b then true else ⟨⟩
λb:Bool.if b then false else ⟨⟩
```

In general:

$$|\Pi x : \mathsf{A}.B(x)| = |B(x_1)| \times |B(x_2)| \times \ldots \times |B(x_{|A|})|$$

$$\text{where } A = \{x_1, x_2, \ldots, x_{|A|}\}$$

$A \leftrightarrow B$ $\log(A)$ $\partial_x A$

Application: reversible programming

$$A \leftrightarrow B$$

Warmup: refactoring with the semiring isomorphisms

The exponent laws tell us

$$a^{1+b} = a^b \times a^1 = a^b \times a$$

and the type `a option` has cardinality `1 + |a|`:

```
type 'a option =
  None : 'a option           (*   1 *)
| Some : 'a -> 'a option     (* + |a| *)
```

So we can turn functions with option arguments into pairs:

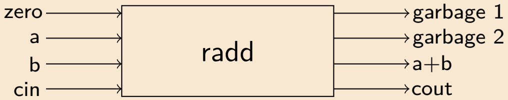
```
let pr : int option -> string = function
  None -> "unknown"
| Some x -> string_of_int x
```

$\rightsquigarrow$

```
let pr : (int -> string) * string =
  (string_of_int, "unknown")
```

Carette & Sabry build **reversible programs** using the equations.

Idea: preserve all input information in the output
— so the program can be run in either direction.



Technique: composition of semiring isomorphisms.

$$\texttt{let adder =}$$
$$\texttt{commut}_\times \circ (\texttt{commut}_\times \otimes \texttt{refl}) \circ \texttt{assoc}_\times \circ \ldots$$

Reversibility, continued

Building block: **isomorphisms**

```
type ('a,'b) iso        (* A ≃ B *)
val refl : ('a,'a) iso  (* A ≃ A *)
val (∘) : ('a,'b) iso -> ('b,'c) iso -> ('a,'c) iso
val symm : ('a,'b) iso -> ('b,'a) iso
```

Building block: **semiring equations**

```
val commut× : ('a *'b, 'b * 'a) iso  (* A×B ≃ B×A *)
```

Building block: **combinators**

```
(* If A ≃ B and C ≃ D then A×C ≃ B×D *)
val (⊗) : ('a,'b) iso -> ('c,'d) iso -> ('a*'c,'b*'d) iso
```

Program by composing **information-preserving transformations**:

```
commut× ∘ (commut× ⊗ refl) ∘ assoc× ∘ ...
```

Application: logarithms of types

$$log(A)$$

Recall: $\log_b(a) = c$ if $b^c = a$. What **type** might $\log(A)$ be?

For types, $\to$ is **exponentiation**.

Pick a free variable α for the base. Then

$$\log_\alpha(A) = C \quad \textit{if} \quad C \to \alpha \;\simeq\; A$$

Equations

$$\log_b(1) = 0 \qquad\qquad b^{\log_b(x)} = x$$
$$\log_b(b) = 1 \qquad\qquad \log_b(b^x) = x$$

$$\log_b(x \times y) = \log_b(x) + \log_b(y)$$
$$\log_b(x^d) = d \times \log_b(x)$$
$$x^{\log_b(y)} = y^{\log_b(x)}$$

Examples

`'a * 'a` $\simeq$ `bool` $\rightarrow$ `'a`:

$$\log_{\alpha}(\alpha \times \alpha) = \log_{\alpha}(\alpha) + \log_{\alpha}(\alpha)$$
$$= 1 + 1$$

`unit` $\simeq$ `zero` $\rightarrow$ `'a`:

$$\log_{\alpha}(1) = 0$$

`'c -> 'b -> 'a` $\simeq$ `'c * 'b` $\rightarrow$ `'a`:

$$\log_{\alpha}(C \rightarrow B \rightarrow \alpha) = C \times \log_{\alpha}(B \rightarrow \alpha)$$
$$= C \times B$$

$\log_{\alpha}(A)$ not defined everywhere, e.g.:

$$\log_{\alpha}(B + C) \quad \text{not defined in general}$$

We've seen `'a * 'a` $\simeq$ `bool` $\to$ `'a`:

```
type 'a apair = bool -> 'a

let mkpair : 'a. 'a apair -> ('a * 'a) =
  fun f -> (f false, f true)
```

Instead of `bool` we might use an indexed type:

```
type (_, _) select =
  Fst : ('a * 'b, 'a) select
| Snd : ('a * 'b, 'b) select
```

The indexed type supports a generalized log for `'a * 'b` :

```
type ('a,'b) abpair = {c: 'c. ('a *'b, 'c) select -> 'c }

let g : type a b. (a, b) abpair -> (a * b) =
  fun {c} -> (c Fst, c Snd)
```

Why are logs of types useful?

Logarithms connect **higher order data** (with $\to$)
with **first-order data** (without $\to$).

Sometimes it's simpler to use higher-order data:
$\log_\alpha(B) \to \alpha$ is a **uniform representation** for B.

Functions can be composed; logarithms can locate the A values

Sometimes it's simpler to use first-order data:
B is a **scrutinizable representation** for $\log_\alpha(B) \to \alpha$.

First-order data can be printed, marshalled and inspected

$$\partial_x A$$

Polynomials and derivatives

The constructors 0, 1, $+$ and $\times$ generate **polynomials** over types.

What might we do with a polynomial? **Differentiate** it!
(w.r.t. a free variable x)

$$
\begin{aligned}
\partial_x x &= 1 \\
\partial_x y &= 0 \quad (y \neq x) \\
\partial_x 0 &= 0 \\
\partial_x 1 &= 0 \\
\partial_x (S + T) &= \partial_x S + \partial_x T \\
\partial_x (S \times T) &= (\partial_x S \times T) + (S \times \partial_x T)
\end{aligned}
$$

Type differentiation behaves like standard calculus: $\partial_x x^3 = 3x^2$.

```
type 'x prod₃ = (* x×x×x = x³ *)
  Prod₃ : 'x * 'x * 'x -> 'x prod₃
```

$$
\begin{aligned}
\partial_x\ (x\ \texttt{prod}_3) &= \partial_x(x \times x \times x) \\
&= (\partial_x x \times x \times x) + (x \times ((\partial_x x \times x) + (x \times \partial_x x))) \\
&= (1 \times x \times x) + (x \times ((1 \times x) + (x \times 1))) \\
&= (x \times x) + (x \times x) + (x \times x)
\end{aligned}
$$

The result is a **context**[2], i.e. the original type with one x removed:

```
type 'x position = (* x² + x² + x² = 3x² *)
  Left  :       'x * 'x -> 'x position
| Mid   : 'x *       'x -> 'x position
| Right : 'x * 'x       -> 'x position
```

[2] *The derivative of a regular type is its type of one-hole contexts* (McBride, 2001)

Derivatives of fixed points & substitutions

Derivatives extend to **fixed points** and **substitutions**:

$$\partial_x\ (\mu y.A) = \mu z.(\partial_x A)[y := \mu y.A] + (\partial_y A)[y := \mu y.A] \times z$$

$$\partial_x(A[y := B]) = (\partial_x A)[y := B] + (\partial_y A)[y := B] \times \partial_x B$$

following a 2-argument variant of the “chain rule”.

Polynomials and derivatives (example: elements in lists)

The derivative of a list is a pair of lists:

$$\begin{aligned}
\partial_x\ (x\ \mathsf{list}) &\simeq \partial_x\ (\mu y.1 + x \times y) \\
&\simeq \mu z.(\partial_x(1 + x \times y))[y := x\ \mathsf{list}] \\
&\quad + (\partial_y(1 + x \times y))[y := x\ \mathsf{list}] \times z \\
&\simeq \mu z.y[y := x\ \mathsf{list}] + x[y := x\ \mathsf{list}] \times z \\
&\simeq (x\ \mathsf{list} + x\ \mathsf{list})
\end{aligned}$$

A pair of lists is the **one-hole context** for a list:

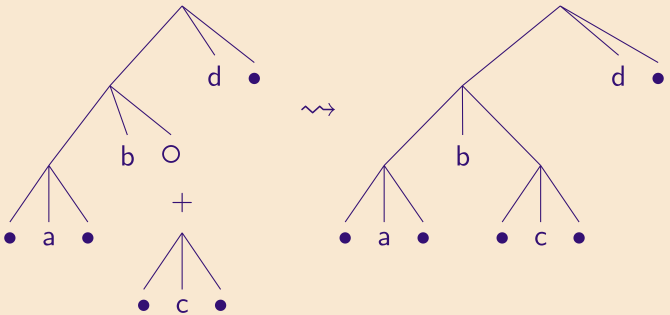
$[\mathtt{x}_0;\ \mathtt{x}_1;\ \mathtt{x}_2;\ \mathtt{x}_3;\ \mathtt{x}_4;\ \mathtt{x}_5;\ \mathtt{x}_6]$

$[\mathtt{x}_0;\ \mathtt{x}_1;\ \mathtt{x}_2]$ $\qquad$ $[\mathtt{x}_4;\ \mathtt{x}_5;\ \mathtt{x}_6]$

Why are derivatives of types useful?

The derivative of a type is its type of **one-hole contexts**.

One-hole contexts are useful for navigating & decomposing trees[3].



[3] *The Zipper* (Huet, 1997)

Next time: monads (etc.)

$\gg=$