FUNCTIONAL PROGRAMMING:

Destination or Origin?

THE ESSENCE OF FUNCTIONAL PROGRAMMING

Definition (Kleisli Category)**.** Given a monad T over a category $\mathcal{C}$, the *Kleisli category* $\mathcal{C}_T$ of T is defined as follows:

- $\mathrm{Ob}(\mathcal{C}_T) = \mathrm{Ob}(\mathcal{C})$
- $\mathrm{Hom}_{\mathcal{C}_T}(A, B) = \mathrm{Hom}_C(A, TB)$
- identity morphisms in $\mathcal{C}_T$ are $\eta_X \in \mathrm{Hom}_{\mathcal{C}_T}(X, X) = \mathrm{Hom}_{\mathcal{C}}(X, TX)$
- composition of $f : A \to TB$ and $g : B \to TC$ is *Kleisli composition*: $g^* f : A \to TC$

Theorem 5. $\mathcal{C}_T$ is a category:

1. $\eta^* f = \mathrm{id} \circ f = f$
2. $f^* \eta = f$

I assume you know this.

JUST KIDDING.

"What's great about Kleisli arrows is that (a) they capture the non-pure notion of 'function' common to other languages while explicitly marking the 'functionoids' (or, to use the proper term, arrows) as occurring in a monad and (b) give the monad laws a very clean, obvious presentation."

–Michael O. Church

When talking about FP, start with this: make the connection to "non-pure notion of "'function'" the *default*. FP has no problem with mutation or I/O whatsoever, and we should be polite but firm in establishing that from the outset.

BUT WAIT, THERE'S MORE!

"Kleisli arrows also help one to group the symbols in a -> m b as a {-> m} b rather than the technically correct but opaque a -> {m b}. From this viewpoint, we come to see the relationship between this Monad (which is itself quite simple; it's just a type class) concept and the notion of 'computational context'. An a -> m b is like a a -> b but is allowed to do m-stuff."

–Michael O. Church

The Kleisli category and Kleisli arrows answer the question "How do you program compositionally with a computational context," which, again, we almost always have. Having an empty context (Id monad) or no context (a -> b) are *special cases*, and we should present them as such.

LOGIC PROGRAMMING: MICROKANREN

```
> (define a-and-b
    (conj
      (call/fresh (λ (a) (≡ a 7)))
      (call/fresh (λ (b) (disj (≡ b 5) (≡ b 6))))))
> (a-and-b empty-state)
((((#(1) . 5) (#(0) . 7)) . 2)
 (((#(1) . 6) (#(0) . 7)) . 2))
```

If you think about it for a second, this is *weird*: it seems like the λ with the `disj` returns *twice*, once with `b` bound to 5, once with `b` bound to 6, but both in the context of the `conj`, where `a` is bound to 7, leading to the two solutions.

THE ESSENCE OF LOGIC PROGRAMMING

```
(define (== u v)
  (lambda (s/c)
    (let ((s (unify u v (car s/c))))
      (if s (unit `(,s . ,(cdr s/c))) mzero))))

(define (unit s/c) (cons s/c mzero))
(define mzero '())

(define (disj g1 g2) (lambda (s/c) (mplus (g1 s/c) (g2 s/c))))
(define (conj g1 g2) (lambda (s/c) (bind (g1 s/c) g2)))

(define (mplus $1 $2)
  (cond
    ((null? $1) $2)
    ((procedure? $1) (lambda () (mplus $2 ($1))))
    (else (cons (car $1) (mplus (cdr $1) $2)))))

(define (bind $ g)
  (cond
    ((null? $) mzero)
    ((procedure? $) (lambda () (bind ($) g)))
    (else (mplus (g (car $)) (bind (cdr $) g)))))
```

The “essence” is using MonadPlus for nondeterminism. If unification succeeds we construct an instance of the MonadPlus, otherwise we fail (`mzero`). Note interleaving of arguments in both `mplus` and `bind`; this is for fair scheduling when one branch or another fails. Note `disj` is just `mplus g1 g2`, i.e. goal 1 or goal 2. `mplus` reminding of sum type (this or that) is not an accident.

RELATIONS AS FUNCTIONS

"We use functions to simulate relations. An arbitrary n-ary relation is viewed as an (n−1)-ary partial function, mapping tuples of domain elements into a linearized submultiset of elements of the codomain over which the initial relation holds. A given collection of goals may be satisfied by zero or more states. The result of a μKanren program is a stream of satisfying states. The stream may be finite or infinite, as there may be finite or infinitely many satisfying states."

–Jason Hemann, Daniel P. Friedman

Partiality represented by `mzero` (failure); choice represented by `mplus`. "Comptational context" is the set of variable bindings and the variable counter.

COMPUTATIONAL CONTEXT

- The "m" in "mzero" and "mplus" means what you think it means.
- Monads carry a computational context through computations.
- Contexts you've probably heard a lot about here: state, I/O, concurrency…
- MonadPlus here captures partiality (failure) and backtracking.
- Monads revealed! They're just patterns of higher-order functions!

Purely-functional μKanren makes nice and clear how simple even MonadPlus is: `mzero` is just the empty list; `unit` is just `cons`. The "magic" is in `bind` and `mplus`, which make list the "nondeterminism monad."

PROBABILISTIC PROGRAMMING

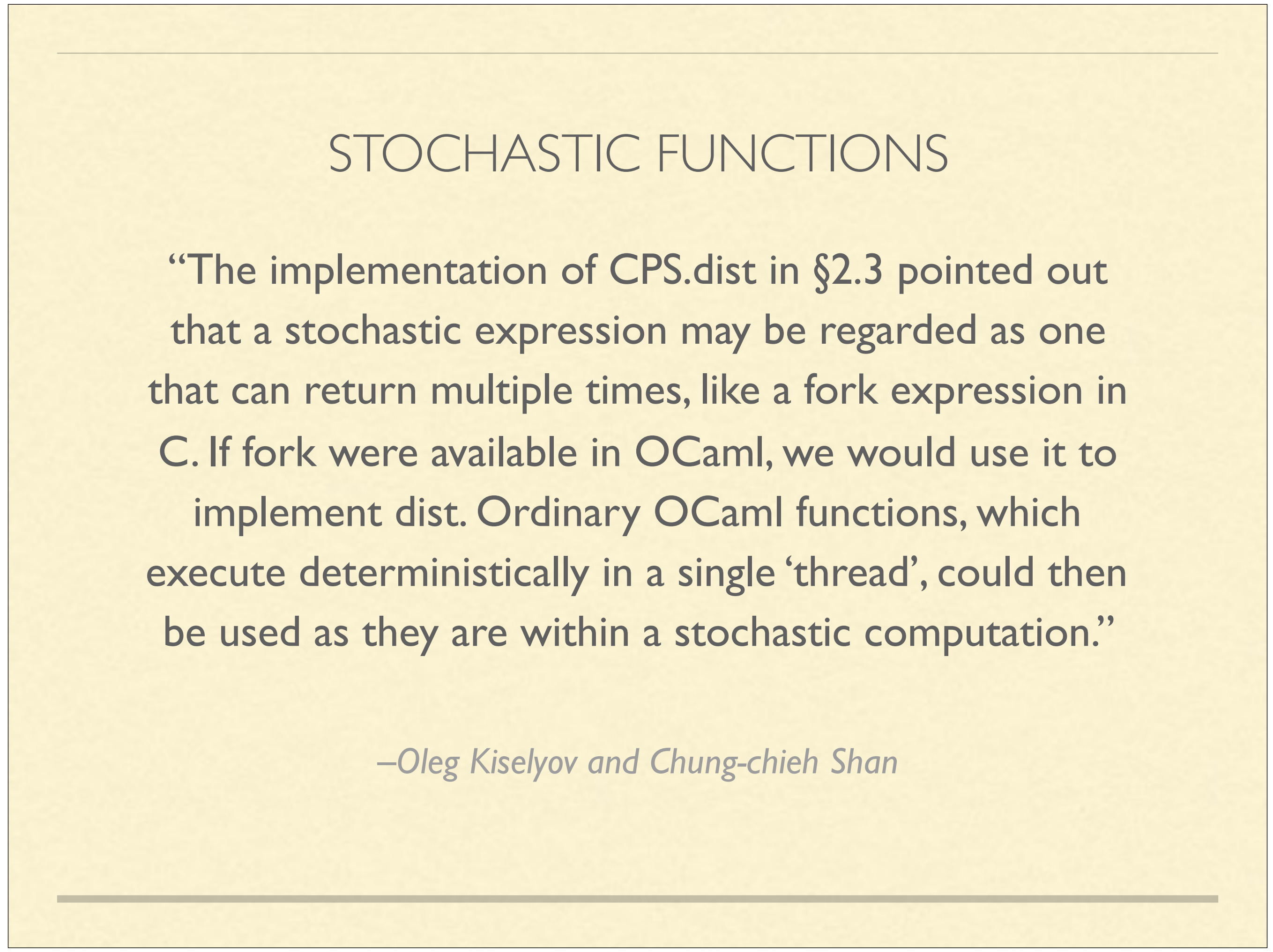
```
let flip = fun p -> dist [(p, true); (1.-.p, false)];;

let grass_model = fun () ->
  let cloudy    = flip 0.5 in
  let rain      = flip (if cloudy then 0.8 else 0.2) in
  let sprinkler = flip (if cloudy then 0.1 else 0.5) in
  let wet_roof  = flip 0.7 && rain in
  let wet_grass = flip 0.9 && rain || flip 0.9 && sprinkler in
  if wet_grass then rain else fail ()
;;

let t1exact = exact_reify grass_model;;
let [(0.4581, V true); (0.188999999999999974, V false)]
    = t1exact;;

let normalize l =
  let total = List.fold_left (fun acc (p,_) -> p +. acc) 0.0 l in
  List.map (fun (p,v) -> (p /. total,v)) l;;

let t1exact' = normalize t1exact;;
let [(0.707927677329624472, V true); (0.292072322670375473, V false)]
    = t1exact';;
```

The standard “wet grass” Bayesian belief network, as a plain OCaml program with an embedded probabilistic DSL. Note `dist` returns a probability distribution, not a Boolean, but we can use plain “if” expressions with it anyway! This again smells like maybe `dist` can return more than once…

ESSENCE OF PROBABILISTIC PROGRAMMING: MONADIC

```
module SearchTree = struct
   type 'a pm = 'a pV
   type ('a,'b) arr = 'a -> 'b pV
   let b = pv_unit
   let dist ch = List.map (fun (p,v) -> (p, V v)) ch
   let neg e = pv_bind e (fun x -> pv_unit (not x))
   let con e1 e2 = pv_bind e1 (fun v1 ->
                    if v1 then e2 else (pv_unit false))
   let dis e1 e2 = pv_bind e1 (fun v1 ->
                    if v1 then (pv_unit true) else e2)
   let if_ et e1 e2 = pv_bind et (fun t ->
                       if t then e1 () else e2 ())
   let lam e = pv_unit (fun x -> e (pv_unit x))
   let app e1 e2 = pv_bind e1 (pv_bind e2)
  end
```

A module expressing the DSL monadically, with the types of values and functions, and value constructor `b`, `dist` for constructing distributions, `neg` for negation, `con` for conjunction, `dis` for disjunction, `if_` for conditionals, `lam` for abstraction, and `app` for function application. *Lots* of syntactic and runtime overhead; client code will be *very* ugly.



Motivating “expressions that can return multiple times.” In probabilistic programming, you want the values returned to have different weights.

ESSENCE OF PROBABILISTIC PROGRAMMING: DIRECT

```
module Direct = struct
   type 'a pm = 'a
   type ('a,'b) arr = 'a -> 'b
   let b x = x
   let dist ch = shift (fun k ->
     List.map (function (p,v) -> (p, C (fun () -> k v))) ch)
   let neg e = not e
   let con e1 e2 = e1 && e2
   let dis e1 e2 = e1 || e2
   let if_ et e1 e2 = if et then e1 () else e2 ()
   let lam e = e
   let app e1 e2 = e1 e2
   let reify0 m = reset (fun () -> pv_unit (m ()))
  end
```

The same probabilistic DSL with delimited continuations. Note the value type is just the type; arrow type is just arrow; value constructor is just identity; etc. `dist` now makes the implicit continuation explicit and thunkifies applying the continuation to the values in the distribution, and `reify0` delimits the continuation at thunkifying constructing the probability monad by applying the probabilistic program to `Unit`, giving us the client code on the example slide.

TAKEAWAYS

- “Computational context” isn’t just for state, I/O, and concurrency.
- Monads provide a computational context.
- Delimited continuations provide a computational context.
- Monads and delimited continuations macro-represent each other.
- Delimited continuations are nicer for developing direct-style embedded DSLs.

If you take away nothing else, take the first point.

RESOURCES

- Lecture Notes on Monad-Based Programming
  - <https://git8.cs.fau.de/redmine/projects/mbprog>
- μKanren: A Minimal Functional Core for Relational Programming
  - <http://webyrd.net/scheme-2013/papers/HemannMuKanren2013.pdf>
- Purely Functional Lazy Nondeterministic Programming
  - <http://okmij.org/ftp/Haskell/FLP/lazy-nondet.pdf>
- Embedded Probabilistic Programming
  - <http://okmij.org/ftp/kakuritu/dsl-paper.pdf>

References, all of which are included in the repository.