The joy of cats in functional programming

Tarmo Uustalu, Institute of Cybernetics

joint with
Varmo Vene, U of Tartu
Thorsten Altenkirch, U of Nottingham
Venanzio Capretta, Radboud U Nijmegen
Robin Cockett, U of Calgary
Neil Ghani, U of Nottingham
Makoto Hamana, U of Tokyo
Ichiro Hasuo, Bart Jacobs, Radboud U Nijmegen
Alberto Pardo, U de la República, Montevideo

Final workshop of CDC, Tallinn, 21–22 Jan. 2008

What is this about?

- Functional programming (in cool languages like in Haskell, OCaml) is about programming with mathematical functions or almost so.
- We believe in mathematical structures in functional programming, both in data and control.
- We believe these structures are older than us, they are there to be discovered rather than invented.
- Moreover, it often amounts to rediscovering what was already known in category theory.
- Your program is not good until it is structured well. Especially if you want to reuse it, show it to a friend or reason about it.
- We believe in no less, believe it or not!
- So we need to care about the right structures.

Category theory

- This is mathematics about categories, functors, natural transformations and the like.
- Related to algebra, but far more general.
- Glasses to see ever-repeating structures clearly.
- You can think of your type and program denotations as living in categories, e.g.,
  sets and functions, in the case of simply typed lambda calculus
  pers, in the case of parametric polymorphism
  cpos, in the case of nontermination from general recursion
- The fun is to see the same thing again and say, hey, I know how this works!
- (Do you see why?)
- In a slightly more syntax-driven mindmode, type theorists are often concerned about the same things as categorical program semanticists.

Haskell "humor"

- *The evolution of a Haskell programmer* by Fritz Ruehr (http://www.willamette.edu/∼fruehr/haskell/evolution.html)
- Freshman Haskell programmer

```
fac n = if n == 0
           then 1
           else n * fac (n-1)
```

- Junior Haskell programmer (beginning Peano player)

```
fac  0    =  1
fac (n+1) = (n+1) * fac n
```

- Senior Haskell programmer (voted for Nixon, Buchanan, Bush, "leans right")

```
fac n = foldr (*) 1 [1..n]
```

- Memoizing Haskell programmer (takes Ginkgo Biloba daily):

```
facs = scanl (*) 1 [1..]

fac n = facs !! n
```

- Post-doc Haskell programmer (from Uustalu, Vene and Pardo's *Recursion Schemes from Comonads*, NJC 2001)

```
-- explicit type recursion with functors and catamorphisms

newtype Mu f = In (f (Mu f))

unIn (In x) = x

cata phi = phi . fmap (cata phi) . unIn

-- base functor and data type for natural numbers

data N c = Z | S c

instance Functor N where
  fmap g  Z    = Z
  fmap g (S x) = S (g x)

type Nat = Mu N

zero   = In  Z
suck n = In (S n)
```

```
add m = cata phi where
  phi  Z     = m
  phi (S f) = suck f

mult m = cata phi where
  phi  Z     = zero
  phi (S f) = add m f
```

```haskell
-- explicit products and their functorial action

data Prod e c = Pair c e

outl (Pair x y) = x
outr (Pair x y) = y

fork f g x = Pair (f x) (g x)

instance Functor (Prod e) where
  fmap g = fork (g . outl) outr

-- comonads, the categorical "opposite" of monads

class Functor n => Comonad n where
  extr :: n a -> a
  dupl :: n a -> n (n a)

instance Comonad (Prod e) where
  extr = outl
  dupl = fork id outr

-- generalized catamorphisms, zygomorphisms and paramorphisms

gcata :: (Functor f, Comonad n) =>
       (forall a. f (n a) -> n (f a)) -> (f (n c) -> c) -> Mu f -> c
gcata dist phi = extr . cata (fmap phi . dist . fmap dupl)

zygo chi = gcata (fork (fmap outl) (chi . fmap outr))

para :: Functor f => (f (Prod (Mu f) c) -> c) -> Mu f -> c
para = zygo In
```

- . . . and finally

```
-- factorial, the *hard* way!

fac = para phi where
  phi  Z                = suck zero
  phi (S (Pair f n)) = mult f (suck n)

-- for convenience and testing

int = cata phi where
  phi  Z     = 0
  phi (S f) = 1 + f

instance Show (Mu N) where
   show = show . int
```

- Tenured professor (teaching Haskell to freshmen)

```
fac n = product [1..n]
```

Less "humorous"

- For less sarkastic expressions of appreciation read, eg,
  `http://www.haskell.org/haskellwiki/Research_papers/Monads_and_arrows`
  `http://www.haskell.org/haskellwiki/Lucid`
  `http://www.haskell.org/haskellwiki/Zipper`
- or
  `http://sigfpe.blogspot.com/2006/06/monads-kleisli-arrows-comonads-and.html`
  and further entries on Dan Piponi (aka Sigfpe's) blog
- or
  related entries on Lambda the Ultimate.
- (To disillusion you: You can't really improve our citation records with TKN by visiting these pages. . .)

Rest of this talk

- Briefly about what we did 2002-07:
- Structured recursion:
  - structured recursion schemes from comonads (ie postdoc programming), recursive coalgebras
  - Mendler recursion, aka type-based termination, aka circular proofs
  - foundations for shortcut deforestation
- Effects and context-dependence:
  - combining monadic effects
  - nontermination as a monadic effect
  - context-dependence via comonads (CDC)

Recursion schemes from comonads (U, Vene, Pardo)

- Recursion in total (terminating/productive) programming, as in sets and functions, is only possible in relation to inductive/coinductive types or families.
- Categorically, inductive types (such as the types of naturals, lists, trees of various flavors etc) are initial algebras of endofunctors (= initial algebras given by signatures in universal algebra).
- The most basic form of recursion (known as iteration in recursion theory, fold in FP) corresponds to the (defining) unique homomorphism property of initial algebras:
  For an endofunctor F with an initial algebra $(\mu F, \mathsf{in}_F)$, we have

$$\begin{array}{ccc} F(\mu F) & \xrightarrow{\mathsf{in}_F} & \mu F \\ {\scriptstyle Ff}\downarrow & & \downarrow{\scriptstyle \mathsf{fold}(\phi) =_{\mathrm{df}} \exists! f} \\ FC & \xrightarrow{\forall \phi} & C \end{array}$$

- We proved this powerful generic function definition scheme, a many-in-one recursion scheme parametrized by a recursive call pattern captured in a comonad and distributive law:
  Given an endofunctor F with an initial algebra $(\mu F, \mathrm{in}_F)$ and a D with a distributive law of F over D, we have

$$
\begin{array}{ccccc}
F(D(\mu F)) & \xleftarrow{F\iota} & F(\mu F) & \xrightarrow{\mathrm{in}_F} & \mu F \\
\Big\downarrow{\scriptstyle F(Df)} & & & & \Big\downarrow{\scriptstyle \exists! f} \\
F(DC) & & \xrightarrow{\quad\quad \forall \phi \quad\quad} & & C
\end{array}
$$

  where $(\mu F, \iota)$ is a specific E-M coalgebra of the comonad, induced by the distributive law. A comonad is an endofunctor with additional data and properties.
- Postdoc factorial is but one example . . . and slightly past the point.
- Beyond primitive recursion, it covers course-of-value recursion, recursion with subsidiary simultaneous recursions on structurally smaller arguments etc.

Recursive coalgebras (Capretta, U, Vene)

- The algebra structure in_F of an initial F-algebra is an isomorphism.
- In fact, recursion is more about its inverse, a coalgebra (a carrier with observations rather than operations)!
- Stepping back and following Osius '70s, we defined any F-coalgebra (A, α) to be recursive (supporting recursion) if it satisfies

$$\begin{array}{ccc} FA & \xleftarrow{\ \alpha\ } & A \\ {\scriptstyle Ff}\big\downarrow & & \big\downarrow{\scriptstyle \exists! f} \\ FC & \xrightarrow{\ \forall \phi\ } & C \end{array}$$

- We identified a number of ways for constructing recursive coalgebras out of coalgebras already known to be recursive.
- These included a construction based on comonads and distributive laws, generalizing "recursion schemes from comonads" to coalgebras other than inverses of initial algebras.

Mendler-style recursion, aka type-based termination, aka circular proofs (U, Vene, Cockett)

- Programming with recursors defined by properties such as initiality, "recursion schemes from comonads" is cumbersome.
- In actual FP, one wants to program with something closer to general recursion, even if it must be well-behaved.
- So our recursors need some fine-tuning to be usable.
- We explored the idea (proposed in type theory by Mendler '87) to induce maps $\mu F \to C$ not by maps $\phi : FC \to C$ but by natural transformations $\Phi_Y : \mathcal{C}(Y, C) \to \mathcal{C}(FY, C)$, for fold. By Yoneda lemma, these are in natural bijection.

- This gives indeed a program construct which behaves (seemingly) similarly to a general recursor.
- Checking conformance of what a priori is a general recursion to the fold scheme becomes type-checking. Instead of just admitting the general recursion typing $\mathcal{C}(\mu F, C) \to \mathcal{C}(F(\mu F), C)$ we require the recursive definition body to admit the more general type $\mathcal{C}(Y, C) \to \mathcal{C}(FY, C)$.
- This extends to other recursion schemes.
- Ultimately, Mendler-style recursion from the cofree recursive comonad is equivalent to what are known as "circular proofs".
- Circular proofs is a codename for proof systems with a notion of proof that accepts progressive infinite paths in proof trees, studied eg by Santocanale, now promoted by Brotherston & Simpson.

Shortcut fusion: build and augment (U, Vene, Ghani)

- Something similar appears in shortcut deforestation, a program transformation for eliminating intermediate datastructures.
- Instead of taking $\mathrm{in}_F : F(\mu F) \to \mu F$ to be the basic means to construct data in μF, one can take an operation known as build to be basic.
- Build is an operation taking a strongly dinatural transformation $\Theta_X : \mathcal{C}(FX \to X) \to \mathcal{C}(A \to X)$ to a map $A \to \mu F$.
- Shortcut fusion is based on this rule: $\mathrm{fold}(\phi : FC \to C) \circ \mathrm{build}(\Theta) = \Theta_C(\phi)$.
- We gave a category-theoretic explanation of build and shortcut fusion in terms of limits of an algebra-structure forgetting functors.
- Moreover, we gave a general monad-based account of what had been ad hoc extension of build, called augment.

Combining monadic effects (Ghani, U)

- It is common in functional programming and mathematical program semantics to abstract effects into monads.
- A monad is a functor (type constructor) together with two natural transformations (polymorphic functions), with some specific properties.
- In particular, if T is a monad on some base category $\mathcal{C}$, it defines a category called the Kleisli category whose objects are those of $\mathcal{C}$ but maps $A \to B$ are maps $A \to TB$ of $\mathcal{C}$.
  Seeing TB as the type of effectful computations of values of B, maps $A \to TB$ become effectful functions. The monad tells what the identities of effectful functions are and how they compose.
- Eg, exception-raising functions $A \to B$ are really maps $A \to B + E$, ie, Kleisli maps for $TB =_{\mathrm{df}} B + E$,
  stateful functions $A \to B$ are really maps $A \times S \to B \times S$.
  These are in bijection with maps $A \to S \Rightarrow B \times S$, which are Kleisli maps for $TB =_{\mathrm{df}} S \Rightarrow B \times S$.

- It is tricky to combine effects.
- Some canonical ways are distributive laws (exists between some monads) and coproduct of monads.
- Computing the coproduct of two comonads is tedious in general (it's nothing like the coproduct of functors, which is computed pointwise).
- We gave a specific construction for ideal monads, ie, monads of the form $TA =_{\mathrm{df}} A + T'A$ with the unit given by left injection and multiplication restricting to T' in an appropriate sense.
- This covers quite a few examples, eg nondeadlocking nondeterminism and probabilistic choice.

Nontermination as a monadic effect rather than defect (Capretta, Altenkirch, U)

- Type-theorists cannot accept that pure functions may fail to terminate.
- Or more exactly, it is free general recursion and even more basically looping that are problematic. (Programs are proofs and you better don't prove anything by general recursion.)
- This can be remedied by paying for loops: computation takes time.
- Nontermination then becomes a monadic effect as any other.
- The monad is $TA =_{\mathrm{df}} \nu X.A + X$ (the final coalgebra of $X \mapsto A + X$, exists in sets) implemented in Haskell by

```
data Delay a = Now a | Later (Delay a)     -- read coinductively

instance Monad Delay where
  return a = Now a
  Now a >>= k = k a
  (Later c) >>= k = Later (c >>= k)
```

- One can define a never-terminating computation and an (unfair) race of two computations:

```
never :: Delay a
never = Later never

race :: Delay a -> Delay a -> Delay a
race (Now a)   c          = Now a
race (Later c) (Now a)    = Now a
race (Later c) (Later c') = Later (race c c')
```

- Further one gets a looping ("iteration") combinator and a general recursor that fit into sets-like settings.
- One can quotient T by equating computations that differ by a finite wait.
- This gives a denotational semantics for languages with general recursion without involving cpos (in fact this is untrue, as the Kleisli category is a cpo-category).
- Moreover, nontermination as an effect thus defined can be combined with other effects in standard ways (distributive laws, coproduct of monads).

Context-dependence via comonads (U, Vene)

- If monads can be used to structure effectful notions of computation, is there a similar use for the formal dual, comonads?
- The answer is yes: comonads capture notions of context-dependence.
- Given a comonad D on a base category $\mathcal{C}$, maps $DA \to B$ of $\mathcal{C}$ are maps $A \to B$ of the coKleisli category, with identities and composition.
  DA can be thought of as the the type of values of A embedded in a context. The coKleisli category is then the category of context-dependent functions.
- For $DA =_{\mathrm{df}} (\mathsf{Nat} \Rightarrow A) \times A$, these maps $(\mathsf{Nat} \Rightarrow A) \times \mathsf{Nat} \to B$ are in bijection with maps $\mathsf{Nat} \Rightarrow A \to \mathsf{Nat} \Rightarrow B$, ie, maps $\mathsf{Str}A \to \mathsf{Str}B$, general stream functions (which can represent discrete-time dignal transformers). The coKleisli identities and composition agree with those of stream functions.

- Causal stream functions are captured by $DA =_{\mathrm{df}} \mathsf{List} A \times A$ with the 1st component for the past of the signal and the 2nd component the present.
- General tree relabellings are captured by trees-with-a-position (zipper) comonad.
- Further examples are eg cellular automata (Game of Life).
- This gives a foundation for a generic denotational semantics of (higher-order) languages for context-dependent computation (dataflow languages à la Lucid, Lustre/Lucid Synchrone, attribute grammars etc).

Conclusion

- Structure abounds in functional computation. It's only to be surfaced and exploited (so it can then be pushed to the background again).
- Here, comonads and monads were central, but this is more of an incident than rule.