

# The Monad of Strict Computation

*A Categorical Framework for the Semantics  
of Languages in which Strict and Non-strict  
computation rules are mixed*

Dick Kieburtz

*Portland State University*

WG2.8 Meeting

July 16-20, 2007

# The Problem, illustrated

- Consider the Haskell datatype:

**data** *Slist* a = Nil | Scons !a (*Slist* a)

- What is an appropriate denotation for *Scons*?
  - *Scons* can be used to define the *seq* function
$$\text{seq } x \ y = \mathbf{case} \ \text{Scons } x \ \text{Nil} \ \mathbf{of} \ \{ \_ \rightarrow y \}$$
  - *Scons* should be modeled by a curried function, but its uncurried equivalent is *not* simply the injection of a cartesian product of types.
- What domain structure models the data type *Slist*?
  - *Slist* can be modeled by a sum, but it's not a sum of products.
- Is there a simple structure with which to characterize a domain for *Slist*?

# Frame Semantics

## (a quick review)

- A *frame* category is a type-indexed, cartesian category,  $\mathcal{D}$ , with the additional structure
  - $\mathcal{D}$  is equipped with a family of operations,

$$\bullet_{\tau} :: \forall \tau'. (D_{\tau' \rightarrow \tau} \times D_{\tau'}) \rightarrow D_{\tau}$$

- A frame category,  $\mathcal{D}$ , is *extensional* if
$$\forall \tau, \tau'. \forall f, g \in D_{\tau' \rightarrow \tau}. (\forall d \in D_{\tau}. f \bullet_{\tau} d = g \bullet_{\tau} d) \Rightarrow f = g$$
- An arrow  $\varphi \in D_{\tau'} \rightarrow D_{\tau}$  is *representable* if
$$\exists f \in D_{\tau' \rightarrow \tau}. \forall d \in D_{\tau'}. \varphi(d) = f \bullet_{\tau} d$$

# Partial-Order Categories

- Objects of the category *CPO* are sets with complete partial orders.
  - A p.o. set is  *$\omega$ -complete* if it contains limits of finite and enumerable chains; *pointed* if it contains a least element.
- Generalize c.p.o. sets to categories
  - Arrows represent  $\sqsubseteq$ , manifesting the order relation
    - Least element,  $\perp$ , of a c.p.o. becomes an initial object in a p-o category
  - Defn: (*Barr & Wells*) A category is said to be  *$\omega$ -cocomplete* if every (small) diagram has a colimit.
  - Characterization of domain objects as partial-order categories is due to
    - *Wand*, 1979, further elaborated by *Smyth-Plotkin*, 1982
- An abstract domain for modeling semantics is a category with products and sums whose objects are  *$\omega$ -cocomplete* categories
  - Its functors preserve order and colimits. (*i.e.* they are *continuous*)
    - Continuous functors are *representable*
    - An  *$\omega$ -cocomplete* frame category is *extensional*
- We take for a semantics domain a *CPO* category,  $\mathcal{D}$ , with all products, an initial object,  $\perp$ , and finite sums
  - $\mathcal{D}$  is  *$\omega$ -cocomplete* (*Smyth-Plotkin*)

# The Monad of Strict Computation

- *Strict* ::  $\mathcal{D} \rightarrow \mathcal{D}$  is analogous to a *Maybe* monad without its explicit data constructors

**data** *Maybe* a = *Nothing* | *Just* a

**monad** *Maybe* **where**

*return* = *Just*

*Nothing* >>= f = *Nothing*

*Just* x >>= f = f x

**monad** *Strict* **where**

*return* = *id*

$\perp$  >>= f =  $\perp$

x >>= f = f x when  $x \neq \perp$

- *Strict* induces a monad transformer, analogous to *MaybeT*

# The tensor product, $\otimes$ , and sum, $\oplus$

- The product in *Strict* becomes a tensor in  $\mathcal{D}$

$$\begin{aligned}(\_,\_)\otimes &:: \text{Strict } a \rightarrow \text{Strict } b \rightarrow \text{Strict } (a \times b) \\(x,y)\otimes &= x \gg= (\lambda x' \rightarrow y \gg= (\lambda y' \rightarrow (x',y')))\end{aligned}$$

- The tensor product has strict projections

$$p_1(x,y)\otimes = x, \quad p_2(x,y)\otimes = y$$

where  $x \neq \perp \wedge y \neq \perp$

$$p_1(x,y)\otimes = p_2(x,y)\otimes = \perp$$

when  $x = \perp \vee y = \perp$

- The sum in *Strict* is a coalesced sum in  $\mathcal{D}$

$$\begin{aligned}inl_{\oplus} &:: \text{Strict } a \rightarrow \text{Strict } (a+b) & inr_{\oplus} &:: \text{Strict } b \rightarrow \text{Strict } (a+b) \\inl_{\oplus} x &= x \gg= (\lambda x' \rightarrow inl x') & inr_{\oplus} y &= y \gg= (\lambda y' \rightarrow inr y')\end{aligned}$$

# The *Lifted* functor

- $Lifted :: \mathcal{D} \rightarrow \mathcal{D}$

$lift :: I \rightarrow Lifted$

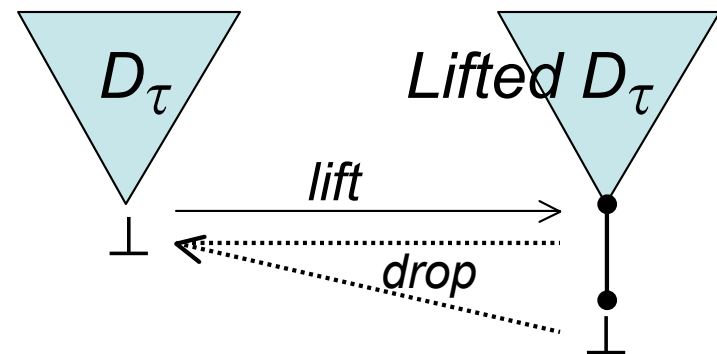
is a natural transformation that injects a pointed type frame,  $D_\tau$  into a domain that adds a new bottom element under  $\perp_\tau$

$drop :: Lifted \rightarrow I$

is the natural transformation that identifies the bottom element of  $Lifted D_\tau$  with the bottom element of  $D_\tau$ .

$drop \circ lift = id$

$lift \circ drop \cong id_{Lifted}$



# The meanings of a data constructor

- A data constructor (of arity  $N$ ) has two formal aspects
  - It maps a sequence of  $N$  types to a new type;
  - It maps  $N$  appropriately typed values to a value in its codomain type
- This suggests its semantic interpretation by a functor
  - Interpretation is in a type-indexed category
    - The object mapping takes  $N$  type frames to another type frame;
    - The arrow mapping takes  $N$  typed arrows (elements of  $N$  type frames) to an arrow (element in the frame of its codomain type)
- An interpretation functor

$[[ \_ ]] :: \text{Type} \rightarrow (\text{tyvar} \rightarrow \text{Strict } \mathcal{D}) \rightarrow \text{Strict } \mathcal{D}$

where  $\text{Type}$  is a “free” category of syntactically well-formed type expressions and compatibly typed term expressions;

$\mathcal{D}$  is a frame category (objects are type frames);

$(\text{tyvar} \rightarrow \text{Strict } \mathcal{D})$  is a type-variable environment.

# Formal semantics of a Haskell data type

- An explicit representation of strictness annotations

**data**  $T a_1 \dots a_m = \dots \mid C (s_1, \gamma_1) \dots (s_n, \gamma_n) \mid \dots$

- Meaning of a strictness annotated type expression

$[[ (s, \gamma) ]] \eta = [[ \gamma ]] \eta$       when  $s = "!"$

$[[ (s, \gamma) ]] \eta = \text{Lifted}([[ \gamma ]] \eta)$       when  $s = ""$

- Meaning of a saturated data constructor application (object mapping)

$[[ C^{(n)} (s_1, \gamma_1) \dots (s_n, \gamma_n) ]] \eta = [[ (s_1, \gamma_1) ]] \eta \otimes \dots \otimes [[ (s_n, \gamma_n) ]] \eta$

- Meaning of a list of alternative type constructions

$[[ \gamma_1 \mid \dots \mid \gamma_p ]] \eta = [[ \gamma_1 ]] \eta \oplus \dots \oplus [[ \gamma_p ]] \eta$

where  $\oplus$  is the sum in category  $\mathcal{D}$  (coalesced bottoms)

- Meaning of a (non-recursive) type constructor declaration

$[[ T a_1 \dots a_m = \gamma ]]_{\text{Decl}} DE \Rightarrow$

$(T = \Lambda \tau_1 \dots \tau_m. [[ \gamma ]] [(a_1 \mapsto \tau_1), \dots, (a_m \mapsto \tau_m)]) \in DE,$

where  $DE$  is a declaration environment

*I've omitted showing data constructor definitions entered into  $DE$*

# Example: a data constructor with strictness annotation

**data**  $S\ a\ b = \dots \mid S1\ !a\ b \mid \dots$

– What's the meaning of the constructor  $S1$ ?

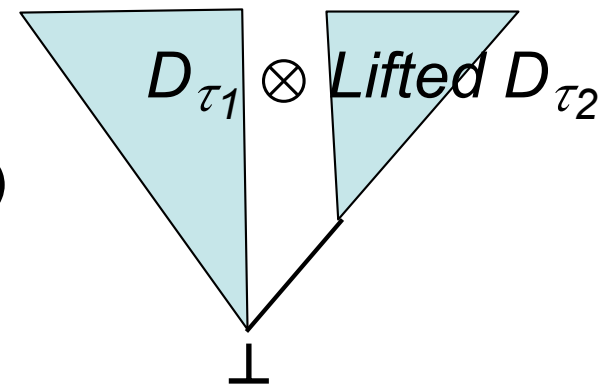
As the object mapping part of a functor:

$$[[ S1 ]] \eta = \Lambda \gamma_1 \gamma_2. [[ \gamma_1 ]] \eta \otimes \text{Lifted}([[\ \gamma_2 ]]) \eta$$

As a data constructor, at a type  $S\ \tau_1\ \tau_2$ :

$$[[ S1 ]]_{Exp} \rho = \lambda x \in D_{\tau_1} y \in D_{\tau_2}. (x, \text{lift } y)_{\otimes}$$

where  $\rho : \text{var}_{\tau} \rightarrow D_{\tau}$  is a typed valuation environment



# Tuple, alternative and arrow types

- Haskell type tuples are lifted products

$$[[(\gamma_1, \gamma_2)]] \eta = \text{Lifted} ([[ \gamma_1 ]] \eta \times [[ \gamma_2 ]] \eta)$$

- Haskell alternatives are coalesced sums

$$[[(\gamma_1 \mid \gamma_2)]] \eta = [[ \gamma_1 ]] \eta \oplus [[ \gamma_2 ]] \eta$$

- Haskell arrow types are lifted encodings of the elements of hom-sets

$$[[(\gamma_1 \rightarrow \gamma_2)]] \eta = \text{Lifted} (\text{code}_{\gamma_1, \gamma_2} (\text{Hom}_{\mathcal{D}} ([[ \gamma_1 ]] \eta, [[ \gamma_2 ]] \eta)))$$

where  $\text{code} :: \text{Hom}(\mathcal{D}) \rightarrow \text{Obj}(\mathcal{D})$  is a bi-natural transformation that codes continuous functions into representations as data

# Semantics of Haskell expressions

$$[[ \_ ]]_{Exp} :: Exp \rightarrow (Var \rightarrow \mathcal{D}) \rightarrow (\mathcal{D} \rightarrow r) \rightarrow r$$

$$[[ e_1 e_2 ]]_{Exp} \rho \kappa = \\ \quad [[ e_1 ]]_{Exp} \rho (\lambda v_1. [[ e_2 ]]_{Exp} \rho (\lambda v_2. \kappa(drop\ v_1 \cdot v_2)))$$

$$[[ \lambda x. e ]]_{Exp} \rho \kappa = \kappa(lift\ (code\ (\lambda v. [[ e ]]_{Exp} \rho [x \mapsto v])))$$

$$[[ (e_1, e_2) ]]_{Exp} \rho \kappa = \\ \quad [[ e_1 ]]_{Exp} \rho (\lambda v_1. [[ e_2 ]]_{Exp} \rho (\lambda v_2. \kappa(lift\ (v_1, v_2))))$$

$$[[ fst ]]_{Exp} \rho \kappa = \kappa(lift\ (\pi_1 \circ drop))$$

$$[[ addInt ]]_{Exp} \rho \kappa = \kappa(lift\ (\lambda x. lift\ (\lambda y. (+)\ (x, y)_{\otimes})))$$

$$[[ C^{(1)} :: (s, \tau) ]]_{Exp} \rho \kappa = \kappa\ lift, \text{ where } s = \text{"!"}$$

$$[[ C^{(1)} :: (s, \tau) ]]_{Exp} \rho \kappa = \kappa\ id, \text{ where } s = \text{""}$$

$$[[ \text{if } e_0 \text{ then } e_1 \text{ else } e_2 ]]_{Exp} \rho \kappa =$$

$$[[ e_0 ]]_{Exp} \rho (\lambda b. b \gg=_{Strict} (\lambda b'. \text{case } b' \text{ of}$$

$$\text{True} \rightarrow [[ e_1 ]]_{Exp} \rho \kappa$$

$$\text{False} \rightarrow [[ e_2 ]]_{Exp} \rho \kappa))$$

# Semantics of Haskell expressions

$[[ \_ ]]_{Exp} :: Exp \rightarrow (Var \rightarrow Strict \mathcal{D}) \rightarrow (\mathcal{D} \rightarrow Strict r) \rightarrow Strict r$

$[[ e_1 e_2 ]]_{Exp} \rho \kappa =$   
 $[[ e_1 ]]_{Exp} \rho >>=_{Strict} (\lambda v_1. [[ e_2 ]]_{Exp} \rho >>=_{Strict} (\lambda v_2. \kappa(drop\ v_1 \cdot v_2)))$

$[[ \lambda x. e ]]_{Exp} \rho \kappa = \kappa (lift (code (\lambda v. [[ e ]]_{Exp} \rho [x \mapsto v])))$

$[[ (e_1, e_2) ]]_{Exp} \rho \kappa =$   
 $[[ e_1 ]]_{Exp} \rho >>=_{Strict} (\lambda v_1. [[ e_2 ]]_{Exp} \rho >>=_{Strict} (\lambda v_2. \kappa (lift (v_1, v_2))))$

$[[ fst ]]_{Exp} \rho \kappa = \kappa (lift (code (\pi_1 \circ drop)))$

$[[ addInt ]]_{Exp} \rho \kappa = \kappa (lift (code (\lambda x. lift (code (\lambda y. x+y))))$

$[[ C^{(1)} :: (s, \tau) ]]_{Exp} \rho \kappa = \kappa (lift (code id)),$  where  $s = "!"$

$[[ C^{(1)} :: (s, \tau) ]]_{Exp} \rho \kappa = \kappa (lift (code lift)),$  where  $s = ""$

$[[ \text{if } e_0 \text{ then } e_1 \text{ else } e_2 ]]_{Exp} \rho \kappa =$

$[[ e_0 ]]_{Exp} \rho >>=_{Strict} (\lambda b. \text{case } b \text{ of}$

$True \rightarrow [[ e_1 ]]_{Exp} \rho \kappa$

$False \rightarrow [[ e_2 ]]_{Exp} \rho \kappa)$

# Recursive Datatype Definitions

## Part 1: Simple recursion; ground types

- Returning to our example, let's substitute for the type parameter:
  - **data**  $Slist\_Int = Nil \mid Scons !Int (Slist\_Int)$
  - Replace the recursive instance on the RHS by a new tyvar
    - **data**  $Slist\_Int = Nil \mid Scons !Int s$   
where  $s = Slist\_Int$
    - The RHS of the declaration is an expression  
 $[s]. Nil \mid Scons !Int s$   
that designates a functor in  $Type \rightarrow Type$
    - Map the expression to the semantic interpretation domain,  
 $\mu$ -binding the variable,  $\zeta$ , which ranges over objects of  $\mathcal{D}$   
$$[[\mu s. Nil \mid Scons !Int s]] \emptyset = \mu \zeta. Lifted\_1 \oplus (D_{Int} \otimes Lifted \zeta)$$
  
which designates the least fixed-point of a functor in  $\mathcal{D} \rightarrow \mathcal{D}$
    - The least fixed-point, computed by iteration, is the meaning of  $Slist\_Int$ , entered into the declaration environment.

# Conclusions

- A categorical framework for semantic domains has some advantages
  - Avoids irrelevant details of representation
  - Dual aspect of a functor (mapping objects & arrows) provides an integrated meaning for constructors
- The *Strict* monad provides a coherent framework in which to model computation rules
  - Simplifies explanation of Haskell's strictness-annotated data constructors
- Simply recursive data types are modeled as initial fixed points of functors that interpret data type declarations
  - An initial fixed point yields an initial algebra in a category of functor algebras
    - Categorical basis for generic programming derivations

End