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

      seq x y = **case** *Scons x Nil* **of** { _ **->** *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,

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

- A frame category, $\mathcal{D}$, is *extensional* if

$$\forall \tau, \tau'. \forall f, g \in D_{\tau'\to\tau}.(\ \forall d \in D_\tau.\ f \bullet_\tau d = g \bullet_\tau d) \Rightarrow f = g$$

- An arrow $\varphi \in D_{\tau'} \to D_\tau$ is *representable* if

$$\exists f \in D_{\tau'\to\tau}.\ \forall d \in D_{\tau'}.\ \varphi(d) = f \bullet_\tau d$$

Partial-Order Categories

- Objects of the category *CPO* are sets with complete partial orders.
  - A p.o. set is ω-*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 ω-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 ω-cocomplete categories
  - Its functors preserve order and colimits. (*i.e.* they are *continuous*)
    - Continuous functors are *representable*
    - An ω-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 ω-cocomplete (*Smyth-Plotkin*)

The Monad of Strict Computation

- *Strict* :: $\mathcal{D} \to \mathcal{D}$ 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 $\mathcal{D}$

$$(\_,\_)_\otimes :: \mathit{Strict}\ a \rightarrow \mathit{Strict}\ b \rightarrow \mathit{Strict}\ (a \times b)$$
$$(x,y)_\otimes = x \mathbin{>\!\!>\!\!=} (\lambda x' \rightarrow y \mathbin{>\!\!>\!\!=} (\lambda y' \rightarrow (x',y')))$$

- The tensor product has strict projections

$$p_1\,(x,y)_\otimes = x, \qquad p_2\,(x,y)_\otimes = y$$
$$\text{where } x \neq \bot \wedge y \neq \bot$$
$$p_1\,(x,y)_\otimes = p_2\,(x,y)_\otimes = \bot$$
$$\text{when } x = \bot \vee y = \bot$$

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

$$\mathit{inl}_\oplus :: \mathit{Strict}\ a \rightarrow \mathit{Strict}\ (a+b) \qquad \mathit{inr}_\oplus :: \mathit{Strict}\ b \rightarrow \mathit{Strict}\ (a+b)$$
$$\mathit{inl}_\oplus\ x = x \mathbin{>\!\!>\!\!=} (\lambda x' \rightarrow \mathit{inl}\ x') \qquad \mathit{inr}_\oplus\ y = y \mathbin{>\!\!>\!\!=} (\lambda y' \rightarrow \mathit{inr}\ y')$$

The *Lifted* functor

- *Lifted* :: $\mathcal{D}$ **->** $\mathcal{D}$

  lift :: *I* $\stackrel{.}{\to}$ *Lifted*

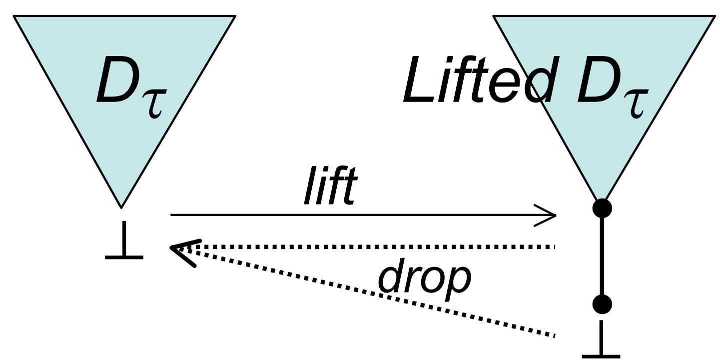
  is a natural transformation that injects a pointed type frame, D_τ into a domain that adds a new bottom element under $\perp_\tau$

  drop :: *Lifted* $\stackrel{.}{\to}$ *I*

  is the natural transformation that identifies the bottom element of *Lifted* D_τ with the bottom element of D_τ.

  drop $\circ$ *lift* = *id*

  lift $\circ$ *drop* $\sqsupseteq$ 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

  $[\![\_]\!] :: \mathrm{Type} \to (\mathit{tyvar} \to \mathit{Strict}\ \mathcal{D}) \to \mathit{Strict}\ \mathcal{D}$

  where *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);

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

Formal semantics of a Haskell data type

- An explicit representation of strictness annotations

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

- Meaning of a strictness annotated type expression

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

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

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

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

- Meaning of a list of alternative type constructions

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

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

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

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

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

  where DE is a declaration environment

  I've omitted showing data constructor definitions entered into DE

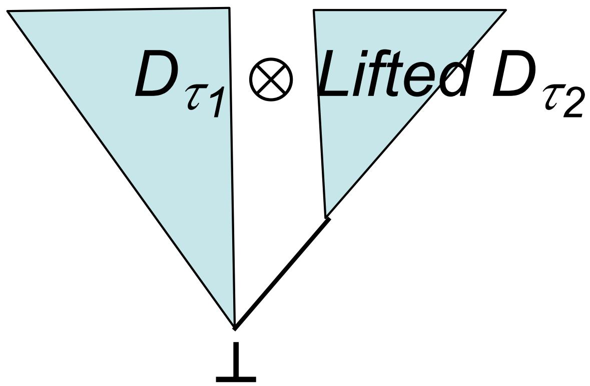
Example: a data constructor with strictness annotation

data *S a b* = … | *S1* !*a* b | …

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

As the object mapping part of a functor:

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



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

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

where $\rho : var_\tau \rightarrow D_\tau$ is a typed valuation environment

Tuple, alternative and arrow types

- Haskell type tuples are lifted products

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

- 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

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

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

Semantics of Haskell expressions

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

$[\![\ e_1\ e_2\ ]\!]_{Exp}\ \rho\ \kappa =$

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

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

$[\![\ (e_1,e_2)\ ]\!]_{Exp}\ \rho\ \kappa =$

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

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

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

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

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

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

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

$\qquad True \to [\![\ e_1\ ]\!]_{Exp}\ \rho\ \kappa$

$\qquad False \to [\![\ e_2\ ]\!]_{Exp}\ \rho\ \kappa))$

Semantics of Haskell expressions

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

$[\![e_1\ e_2]\!]_{Exp}\ \rho\ \kappa =$

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

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

$[\![(e_1,e_2)]\!]_{Exp}\ \rho\ \kappa =$

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

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

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

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

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

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

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

$\qquad True \to [\![e_1]\!]_{Exp}\ \rho\ \kappa$

$\qquad False \to [\![e_2]\!]_{Exp}\ \rho\ \kappa))$

Recursive Datatype Definitions
Part 1: Simple recursion; ground types

- Returning to our example, let's substitute for the type parameter:

  data *Slist_Int* = *Nil* | *Scons* !*Int* (*Slist_Int*)

  – Replace the recursive instance on the RHS by a new tyvar

  data *Slist_Int* = *Nil* | *Scons* !*Int s*

  where *s* = *Slist_Int*

  – The RHS of the declaration is an expression

  [*s*]. *Nil* | *Scons* !*Int s*

  that designates a functor in *Type* → *Type*

  – Map the expression to the semantic interpretation domain, μ-binding the variable, ζ, which ranges over objects of $\mathcal{D}$

  $[[\, \mu s.\ Nil \mid Scons\ !Int\ s]]\ \varnothing = \mu\zeta.\ Lifted\_1 \oplus (D_{Int} \otimes Lifted\ \zeta)$

  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