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:

```haskell
data Slist a = Nil | Scons !a (Slist a)
```

– What is an appropriate denotation for `Scons`?
  • `Scons` can be used to define the `seq` function
    ```haskell
    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 $\omega$-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, $\bot$, 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, $\bot$, and finite sums
  - $\mathcal{D}$ is $\omega$-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
  
  ```haskell
  data Maybe a = Nothing | Just a
  monad Maybe where
  return = Just
  Nothing >>= f = Nothing
  Just x >>= f = f x
  ```

  ```haskell
  monad Strict where
  return = id
  ⊥ >>= f = ⊥
  x >>= f = f x when x ≠ ⊥
  ```

- **Strict** induces a monad transformer, analogous to *MaybeT*
The tensor product, $\otimes$, and sum, $\oplus$

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

\begin{align*}
(\_\_\_\_)_{\otimes} & \:: \text{Strict} \ a \to \text{Strict} \ b \to \text{Strict} \ (a \times b) \\
(x,y)_{\otimes} & = x \gg y = (\lambda x' \to y \gg y = (\lambda y' \to (x',y'))
\end{align*}

- The tensor product has strict projections

\begin{align*}
p_1(x,y)_{\otimes} & = x, & p_2(x,y)_{\otimes} & = y \\
\text{where } x & \neq \bot \land y \neq \bot
\end{align*}

\begin{align*}
p_1(x,y)_{\otimes} & = p_2(x,y)_{\otimes} = \bot \\
\text{when } x & = \bot \lor y = \bot
\end{align*}

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

\begin{align*}
inl_{\oplus} & :: \text{Strict} \ a \to \text{Strict} \ (a+b) & \text{inr}_{\oplus} & :: \text{Strict} \ b \to \text{Strict} \ (a+b) \\
inl_{\oplus} x & = x \gg (\lambda x' \to \text{inl} \ x') & \text{inr}_{\oplus} y & = y \gg (\lambda y' \to \text{inr} \ y')
\end{align*}
The \textit{Lifted} functor

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

  $\text{lift} :: I \rightarrow \text{Lifted}$

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

  $\text{drop} :: \text{Lifted} \rightarrow I$

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

  $\text{drop} \circ \text{lift} = \text{id}$

  $\text{lift} \circ \text{drop} \sqsubseteq \text{id}_{\text{Lifted}}$

\[
\begin{tikzpicture}
  \node (D) at (0,0) {$D_\tau$};
  \node (L) at (3,0) {\text{Lifted} $D_\tau$};
  \draw[->,dashed] (D) to node [left] {$\text{drop}$} (L);
  \draw[->] (L) to node [right] {$\text{lift}$} (D);
\end{tikzpicture}
\]
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
  \[
  [[\_\_]] :: \text{Type} \to (\text{tyvar} \to \text{Strict } D) \to \text{Strict } D
  \]
  where \text{Type} is a “free” category of syntactically well-formed type expressions and compatibly typed term expressions;
  \( D \) is a frame category (objects are type frames);
  \( (\text{tyvar} \to \text{Strict } D) \) is a type-variable environment.
Formal semantics of a Haskell data type

- An explicit representation of strictness annotations
  
  ```
  data T a_1 ... a_m = ... | C (s_1, γ_1) ... (s_n, γ_n) | ...
  ```

- Meaning of a strictness annotated type expression
  
  ```
  [[ (s, γ) ]] η = [[ γ ]] η  \quad \text{when } s = "!"
  [[ (s, γ) ]] η = Lifted([[ γ ]] η)  \quad \text{when } s = ""
  ```

- Meaning of a saturated data constructor application (object mapping)
  
  ```
  [[C^{(n)} (s_1, γ_1) ... (s_n, γ_n) ]] η = [[(s_1, γ_1)]] η ⊗ ... ⊗ [[(s_n, γ_n)]] η
  ```

- Meaning of a list of alternative type constructions
  
  ```
  [[ γ_1 | ... | γ_p ]] η = [[ γ_1 ]] η ⊕ ... ⊕ [[ γ_1 ]] η
  ```
  where ⊕ is the sum in category $D$ (coalesced bottoms)

- Meaning of a (non-recursive) type constructor declaration
  
  ```
  [[ T a_1 ... a_m = γ ]]_{Decl} DE \Rightarrow 
  (T = \Lambda \tau_1...\tau_m. [[ γ ]] [(a_1\mapsto\tau_1), ..., (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

```haskell
data S a b = ... | S1 !a b | ...
```

- What’s the meaning of the constructor \( S1 \)?

As the object mapping part of a functor:

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

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

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

where \( \rho : \text{var}_\tau \to D_\tau \) is a typed valuation environment
Tuple, alternative and arrow types

- **Haskell type tuples are lifted products**
  \[
  [[ (\gamma_1, \gamma_2)]] \eta = \text{Lifted} \left( [[\gamma_1]] \eta \times [[\gamma_2]] \eta \right)
  \]

- **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**
  \[
  [[ (\gamma_1 \rightarrow \gamma_2)]] \eta = \text{Lifted} \left( \text{code}_{\gamma_1, \gamma_2} \left( \text{Hom}_D([[\gamma_1]] \eta, [[\gamma_2]] \eta) \right) \right)
  \]
  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 → (Var → D) → (D → r) → r
[[ e_1 e_2 ]]_{Exp} ρ ι =
  [[ e_1 ]]_{Exp} ρ (λv_1. [[ e_2 ]]_{Exp} ρ (λv_2. ι(drop v_1 • v_2)))
[[ λx.e ]]_{Exp} ρ ι = ι(lift (code (λv. [[ e ]]_{Exp} ρ[x → v])))
[[ (e_1,e_2) ]]_{Exp} ρ ι =
  [[ e_1 ]]_{Exp} ρ (λv_1. [[ e_2 ]]_{Exp} ρ (λv_2. ι(lift (v_1,v_2))))
[[ fst ]]_{Exp} ρ ι = ι(lift (π_1 ° drop))
[[ addInt ]]_{Exp} ρ ι = ι(lift (λx. lift (λy. (+) (x,y)⊗)))
[[ C^{(1)} :: (s,τ) ]]_{Exp} ρ ι = ι lift, where s = “!”
[[ C^{(1)} :: (s,τ) ]]_{Exp} ρ ι = ι id, where s = “”
[[ if e_0 then e_1 else e_2 ]]_{Exp} ρ ι =
  [[ e_0 ]]_{Exp} ρ (λb. b >>=_{Strict} (λb’. case b’ of
  True → [[ e_1 ]]_{Exp} ρ ι
  False → [[ e_2 ]]_{Exp} ρ ι))
Semantics of Haskell expressions

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

\[[ e_1 \, e_2 \] \]_\text{Exp} \rho \kappa =
\begin{align*}
&\left(\left[ e_1 \right] \right)_\text{Exp} \rho \gg=_{\text{Strict}} (\lambda v_1. \left[ e_2 \right] \text{Exp} \rho \gg=_{\text{Strict}} (\lambda v_2. \kappa(\text{drop } v_1 \cdot v_2))) \\
&\left[ \lambda x. e \right] \text{Exp} \rho \kappa = \kappa (\text{lift } (\text{code } (\lambda v. \left[ e \right] \text{Exp} \rho [x \mapsto v]))) \\
&\left[ (e_1,e_2) \right] \text{Exp} \rho \kappa =
\left(\left[ e_1 \right] \right)_\text{Exp} \rho \gg=_{\text{Strict}} (\lambda v_1. \left[ e_2 \right] \text{Exp} \rho \gg=_{\text{Strict}} (\lambda v_2. \kappa (\text{lift } (v_1,v_2)))) \\
&\left[ \text{fst} \right] \text{Exp} \rho \kappa = \kappa (\text{lift } (\text{code } (\pi_1 \circ \text{drop}))) \\
&\left[ \text{addInt} \right] \text{Exp} \rho \kappa = \kappa (\text{lift } (\text{code } (\lambda x. \text{lift } (\text{code } (\lambda y. x+y))))) \\
&\left[ \text{C}^{(1)} :: (s,\tau) \right] \text{Exp} \rho \kappa = \kappa (\text{lift } (\text{code } \text{id})), \quad \text{where } s = "!" \\
&\left[ \text{C}^{(1)} :: (s,\tau) \right] \text{Exp} \rho \kappa = \kappa (\text{lift } (\text{code } \text{lift})), \quad \text{where } s = "" \\
&\left[ \text{if } e_0 \text{ then } e_1 \text{ else } e_2 \right] \text{Exp} \rho \kappa =
\left(\left[ e_0 \right] \right)_\text{Exp} \rho \gg=_{\text{Strict}} (\lambda b. \text{case } b \text{ of}
\hspace{1cm} \text{True } \rightarrow \left[ e_1 \right] \text{Exp} \rho \kappa \\
\hspace{1cm} \text{False } \rightarrow \left[ e_2 \right] \text{Exp} \rho \kappa)\)
Recursive Datatype Definitions
Part 1: Simple recursion; ground types

• Returning to our example, let’s substitute for the type parameter:

```haskell
data Slist_Int = Nil | Scons !Int (Slist_Int)
```

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

```haskell
data Slist_Int = Nil | Scons !Int s

where s = Slist_Int
```

– The RHS of the declaration is an expression

```haskell
[s]. Nil | Scons !Int s
```

that designates a functor in \( \text{Type} \to \text{Type} \)

– Map the expression to the semantic interpretation domain, \( \mu \)-binding the variable, \( \zeta \), which ranges over objects of \( D \)

```haskell
[[ \mu s. Nil | Scons !Int s] \emptyset = \mu \zeta. \text{Lifted}_1 \oplus (\text{D}_{\text{Int}} \otimes \text{Lifted } \zeta)
```

which designates the least fixed-point of a functor in \( D \to D \)

– The least fixed-point, computed by iteration, is the meaning of \( \text{Slist}_\text{Int} \), 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