Lecture 4:

Game models and sequential computability

John Longley
University of Edinburgh

Semantically inspired language design

In denotational semantics, we establish connections:

$$\text{programming language } \mathcal{L} \quad \longleftrightarrow \quad \text{model } \mathcal{M}$$

E.g. full abstraction, definability.

Can start from $\mathcal{L}$ and look for $\mathcal{M}$ to match,
or start from $\mathcal{M}$ and look for $\mathcal{L}$ to match.

The latter approach might lead to:

- interesting new language constructs,
- languages amenable to reasoning.

Moreover, definability ensures we're getting 'value for money' from our chosen $\mathcal{M}$.

Where should we look for models?

Even natural models may lead to unnatural languages!
(Consider PCF+por+exists, or even worse, PCF+H.)

But things are more hopeful for models that can represent a more realistic range of computational phenomena, e.g. game models.

In much work on game semantics to date, the language-driven approach has been dominant. Here we supplement this with some contributions from a model-driven perspective.

Idea: Look at some simple and natural (but structurally rich) game models, and identify the corresponding languages.

Overview, part 1

- Describe the category $\mathcal{G}$ of sequential data structures: models $\otimes$, -o , &.
- Describe three (known) '!' comonads on $\mathcal{G}$, each yielding a model of Intuitionistic Linear Logic. Informally, these embody different notions of 'reusability'.

- This gives us 7 'models':

  $$\mathcal{G} \quad (\mathcal{G}, !_1) \quad (\mathcal{G}, !_2) \quad (\mathcal{G}, !_3) \quad \mathcal{G}_{!_1} \quad \mathcal{G}_{!_2} \quad \mathcal{G}_{!_3}$$

  which we view as models for (an affine version of) Plotkin's FPC, e.g. with types

  $$\tau \ ::= \ \alpha \mid \tau \texttt{*} \tau \mid \tau \texttt{+} \tau \mid \tau \texttt{-o} \tau \mid \texttt{!}\tau \mid \texttt{rectype}\, \alpha \texttt{=>} \tau$$

  (Actually, we use the corresponding 'Fam' categories for call-by-value, as in Abramsky/McCusker 1998.)

Overview, part 2

For each model, we identify an extension of (affine) FPC that defines all computable strategies of suitable type. This leads to:

- a more operational insight into the computational power of each model;
- a selection of language primitives for state encapsulation, coroutining, and backtracking. These are closely related to, but somewhat different from, familiar language constructs.

We give an example to suggest that these primitives may have some practical interest as programming language constructs. (This is currently being explored as part of the Eriskay project.)

Discovering suitable languages

We note that the type

$$\upsilon \quad \equiv \quad \texttt{rectype t => nat -o (nat * t)}$$

denotes a universal object in $\mathcal{G}$, and that all computable strategies of $[\![\,\upsilon\,]\!]$ are affine FPC definable.

So it suffices that a handful of types be programmable retracts of υ, e.g.

$$\upsilon \,\texttt{*}\, \upsilon \qquad \upsilon \,\texttt{+}\, \upsilon \qquad \upsilon \,\texttt{-o}\, \upsilon \qquad \texttt{!}\upsilon$$

The attempt to program these retractions led us, fairly naturally, to our selection of language primitives.

(Here we just present the results of this investigation.)

The category $\mathcal{G}$ (Lamarche 1992, Curien 1993, Abramsky/Jagadeesan 1992)

Write $\mathsf{Alt}(X, Y)$ for the set of finite sequences $z_0 \ldots z_n$ where $z_i \in X$ for i even, $z_i \in Y$ for i odd.

- A game G consists of countable sets O_G, P_G, plus a non-empty prefix-closed set $L_G \subseteq \mathsf{Alt}(O_G, P_G)$ of *legal positions*.

- A strategy for G is a partial function $f : L_G^{odd} \rightharpoonup P_G$ such that $f(s) = y$ implies $sy \in L_G$, and $syx \in \mathrm{dom}\, f$ implies $f(s) = y$.

Games $G \otimes H$, $G \multimap H$ are defined as usual in game semantics. Objects of $\mathcal{G}$ are games; morphisms $G \to H$ are strategies for $G \multimap H$.

Exponential 1: non-repetitive backtracking (Lamarche 1992, Curien 1993, Abramsky/Jagadeesan 1992)

Define $!_1 G$ as follows. Move sets are:

$$O_{!_1 G} \;=\; L_G^{even} \times O_G \qquad\qquad P_{!_1 G} \;=\; P_G$$

Legal positions are sequences $s \in \mathsf{Alt}(O_{!_1 G}, P_{!_1 G})$ such that

- if an O-move (t, x) appears in s then $tx \in L_G$, and if (t, x) is immediately followed by y then $txy \in L_G$;
- if (txy, z) appears in s, then $(t, x)y$ appears earlier in s;
- no O-move (t, x) appears more than once in s.

Intuition: Each O-move backtracks to a previously encountered position in G and explores a new part of the game tree for G.

The co-Kleisli model $G_{!_1}$ is the world of sequential algorithms.

Exponential 2: repetitive non-backtracking (Hyland 1997)

Define $!_2 G$ as follows. Move sets are:

$$O_{!_2 G} \;=\; \mathbb{N} \times O_G \qquad\qquad P_{!_2 G} \;=\; \mathbb{N} \times P_G$$

Legal positions are sequences $s \in \mathsf{Alt}(O_{!_2 G}, P_{!_2 G})$ such that

- for each $i \in \mathbb{N}$, the evident 'subsequence' $s_i \in L_G$;
- if $(i+1, z)$ appears in s, some (i, x) appears earlier in s.

Intuition: Here O can restart a play of G in a fresh 'copy'.
A strategy for $!_2 G$ may behave quite differently in different copies of G (so we get stateful behaviour).

Exponential 3: repetitive backtracking (Harmer/Hyland/Melliès 2007; cf. Longley 2002)

Define $!_3 G$ as follows. Move sets are:

$$O_{!_3 G} \;=\; \mathbb{N} \times O_G \qquad\qquad P_{!_3 G} \;=\; \mathbb{N}_1 \times P_G$$

Legal positions are sequences

$$s \;=\; (a_1, x_1)(b_1, y_1)(a_2, x_2)(b_2, y_2)\ldots \;\in \mathsf{Alt}(O_{!_3 G}, P_{!_3 G})$$

such that

- Each $a_i < i$ and each $b_i = i$. Hence each O-move is either initial (tagged with 0) or points to an earlier P-move.
- For each prefix s' of s, the *thread* (a.k.a. justification sequence) extracted from s' by "chasing pointers" is in L_G.

State encapsulation (cf. Wolverson's thesis)

In $(\mathcal{G}, !_2)$, we can naturally model a language primitive:

$$\text{encaps} : \ (\sigma \ \texttt{*} \ !(\sigma \texttt{*} \tau \ \texttt{-o} \ \sigma \texttt{*} \tau')) \ \texttt{-o} \ !(\tau \ \texttt{-o} \ \tau') \qquad (\tau \text{ ground})$$

This is intermediate in expressive power between first-order and full higher-order store.

If σ is reusable, can also allow non-ground τ, but we then need a syntactic restriction on the "method implementation".

For $!_3$, we should replace $\sigma \texttt{*} \tau'$ by $\sigma \texttt{*} !\tau'$. Also, $!_3$ supports encapsulators for more general imperative style methods, e.g.

$$\text{imp-encaps} : \ (\sigma \ \texttt{*} \ !(\mathrm{RW}\,(\sigma) \ \texttt{-o} \ \tau \ \texttt{-o} \ !\tau')) \ \texttt{-o} \ !(\tau \ \texttt{-o} \ \tau')$$

where $\mathrm{RW}\,(\sigma) = !(\texttt{unit -o } \sigma) \texttt{ * } !(\sigma \texttt{ -o unit})$.

Coroutining operations (cf. Laird 2007)

The bare model $\mathcal{G}$ supports a "resumable exception" operator

$$\begin{aligned} \text{lincatchcont}: \quad & ((\rho \multimap \sigma) \multimap (\tau * \tau')) \quad \multimap \\ & (\tau * ((\rho \multimap \sigma) \multimap \tau') \quad + \quad \rho * (\sigma \multimap (\tau * \tau'))) \end{aligned}$$

where ρ and τ are ground.
In $(\mathcal{G}, !_2)$, for instance, we can improve this to

$$\begin{aligned} \text{catchcont}: \quad & (!(\rho \multimap \sigma) \multimap (\tau * \tau')) \quad \multimap \\ & (\tau * (!(\rho \multimap \sigma) \multimap \tau') \quad + \quad \rho * (\sigma \multimap !(\rho \multimap \sigma) \multimap (\tau * \tau'))) \end{aligned}$$

whence in $\mathcal{G}_{!_2}$ we have an operator

$$\text{coroutine}: \quad ((\rho \to \sigma) \to \rho') \quad \to \quad (\rho \to (\sigma \to \rho) \to \sigma') \quad \to \quad \rho' + \sigma'$$

where $\rho, \sigma, \rho', \sigma'$ are ground.

(Remark: it seems coroutine isn't definable from callcc.)

A 'backtracking' operator

With $!_1$ or $!_3$ (but not $!_2$), we can model an operator

$$\text{force} : \ !(\mathtt{unit} \multimap \tau) \multimap\ !\tau$$

Intuitively, this lets us run some initial phase of a computation once only, and then re-use the results of this phase many times. (Note: the same strategy is also needed to "lift" $\perp$ to $\mathcal{G}_!$.)

For $!_3$, combining force with catchcont, we get

$$\begin{aligned}\text{catchcopy} : \quad & (!(\rho \multimap \sigma) \multimap (\tau * \tau')) \ \multimap \\ & (\tau * (!(\rho \multimap \sigma) \multimap \tau') \ + \ \rho * !(\sigma \multimap !(\rho \multimap \sigma) \multimap (\tau * \tau')))\end{aligned}$$

For $!_1$, we have a "memoizing" variant of this.

The results, part 1

Write AFPC and AEFPC for the affine and affine-exponential variants of FPC. The operators share and switch do (relatively) boring things.

- AFPC+lincatchcont+share is complete for $\mathcal{G}$.
- AEFPC+lincatchcont+share+force+switch is complete for $(\mathcal{G}, !_1)$.
- AEFPC+lincatchcont+encaps is complete for $(\mathcal{G}, !_2)$.
- AEFPC+lincatchcont+encaps+force is complete for $(\mathcal{G}, !_3)$.

The results, part 2

For the co-Kleisli models, we have:

- FPC+memocatchcopy is complete for $\mathcal{G}_{!_1}$.
  (By Cartwright/Curien/Felleisen 1992, so is FPC+catch, but our language seems to promise a tighter "intensional" correspondence.)

- FPC+coroutine (with CBN interpretation) is complete for $\mathcal{G}_{!_2}$ (cf. Laird 2007).

- FPC+catchcopy is complete for $\mathcal{G}_{!_3}$.

A programming application: 'generic search'

Fix $n \in \mathbb{N}$, and consider the problem of counting the vectors $x \in \{0,1\}^n$ satisfying a certain property P. Let's try to do this uniformly in P.

Represent vectors x using `nat -> bool`, and properties P using `(nat -> bool) -> bool`.

Use catchcopy to jump out whenever P requests a new component x_i of x, and then (recursively) resume with $x_i = 0$ and $x_i = 1$ in turn, without having to repeat the computation of $P(x)$ up to this point.

(Also, if P completes having only requested r components, we take care of 2^{n-r} vectors at once.)

Conclusions and further work

Our semantic approach highlights:

- a selection of programming primitives related to, but not the same as, those usually found in existing typed languages;
- certain combinations of these primitives that may coexist without losing runtime safety.

Other issues:

- Connections with op sem: studied in Wolverson's thesis for encaps; more to do for coroutining and backtracking.
- Mathematical relationships between $!_1, !_2, !_3$ are of interest.
- Carry out similar work e.g. for locally non-alternating games.