Eriskay: a programming language based on game semantics

John Longley
Laboratory for Foundations of Computer Science
University of Edinburgh

Joint work with Nicholas Wolverson

An experiment in semantically motivated language design.

Idea: Take some simple and beautiful mathematical model of computation, and design a programming language to fit it.

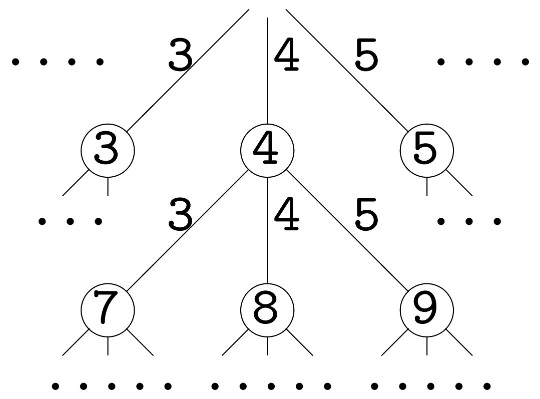
LCF, domain theory (Scott, 1969)	$\longrightarrow$	SML, functional fragment (Milner et al.)
Simple game model (Lamarche, 1992)	$\longrightarrow$	Eriskay (higher-order OO language)

Goal: Do better than SML, using more recent semantic ideas!

The semantic paradigm: *Programs Are Strategies*

(Compare SML, Haskell: *Programs Are Functions*.)

A strategy for a simple Java-style object:



```
Object adder = new Object() {
   private int total = 0;
   public int add (int i) {
    total += i; return total
    }
   }
```

More complex types $\Rightarrow$ more complex games!

Game semantics fits naturally with OO thinking, in that it embodies both data abstraction and a reactive view of computation.

Motivations

- Clean semantic basis aids reasoning about programs, e.g.
  - traditional program correctness (input/output relations)
  - temporal properties of objects
  - observational equivalences (e.g. for class implementations).
- Improved safety properties. E.g. get better static control of exceptions / continuations / name generation in the presence of higher-order store.
- Powerful new programming constructs, e.g. primitives for coroutining and backtracking; higher order class constructs.

‘Real’ or ‘toy’ language?

Eriskay is intended as

- a showcase for semantically inspired language innovations
- a vehicle for realistic programming experiments
- a platform for research in program verification.

‘Full feature’, but mathematical purity is not compromised!

We identify a sublanguage called Lingay, which:

- contains most of the innovative new features
- is suitable for metatheoretical study
- omits polymorphism, type inference, other mod cons.

Where we've got to . . .

- Formal definition of Lingay complete (41 pages), including heap-based operational semantics.

- Working implementation of Lingay available online (a few gaps just now).

- Denotational semantics based on games. (Implementation is a direct animation of this: strategies $\sim$ lazy trees.)

- Proofs of adequacy, definability, full abstraction: substantial ingredients now in place, more to come.

A simple game model (Lamarche, Curien)

Write $Alt(X, Y)$ for the set of finite alternating sequences $x_0 y_0 x_1 y_1 \cdots$

A game G consists of

- disjoint countable sets O_G, P_G of opponent and player moves,
- a non-empty prefix-closed set $L_G \subseteq Alt(O_G, P_G)$ of legal plays.

A strategy for G is a function $f : L_G^{odd} \rightharpoonup P_G$ such that

- if $f(s) = y$ then $sy \in L_G$,
- if $syx \in \mathrm{dom}\, f$ then $s \in \mathrm{dom}\, f$.

Roughly, types are modelled by games, and terms by strategies.

(Actually, for CBV, types are modelled by set-game pairs (A, G).)

Structure in the Lamarche model

It's easy to interpret the Linear Logic connectives $\otimes, \multimap, \&$ as constructions on games.

For ! there are several choices. We work with a rather generous '!' embodying repetitive backtracking. Weaker !'s correspond to sublanguages of Lingay that admit simpler reasoning principles.

The key to modelling stateful behaviour is the contraction strategy of type $!G \multimap !G \otimes !G$.

We can also model much more, e.g.

- recursion at term and type levels,
- subtyping and polymorphism (a bit more than $F_{<:}$).

Interpretation of objects

The type of a Lingay object is essentially its ‘interface’:

$$\{m_1 : \rho_1 \texttt{->} \rho'_1, \;\; \cdots \;\; m_n : \rho_n \texttt{->} \rho'_n\}$$

As proposed in Abramsky et al 1998, we model this by the game

$$\bigotimes_i \, !(\llbracket \rho_i \rrbracket \multimap \llbracket \rho'_i \rrbracket_\perp)$$

Rather like trace semantics (Jeffrey/Rathke, Abrahám/Steffen), but compositional.

At the heart of our approach is a semantic treatment of data abstraction, whereby a strategy of the above type is constructed from a representing strategy, e.g. of type

$$\llbracket \sigma \rrbracket \; \otimes \; \bigotimes_i \, !((\llbracket \rho_i \rrbracket \otimes \llbracket \sigma \rrbracket) \; \multimap \; (\llbracket \rho'_i \rrbracket \otimes \llbracket \sigma \rrbracket)_\perp)$$

Types in (core) Lingay

$$\begin{array}{rcl} \tau & ::= & \texttt{int} \mid \texttt{bool} \mid \tau_1 \multimap \tau_2 \mid\ !\tau \\ & \mid & \{k_1 : \tau_1, \ldots, k_n : \tau_n\} \\ & \mid & \texttt{[|}k_1 : \tau_1, \ldots, k_n : \tau_n\texttt{|]} \\ & \mid & \texttt{rectype}\ t \Rightarrow \tau \\ & \mid & \texttt{classimpl}\ \tau_f, \tau_m, \tau_k\ \texttt{end} \end{array}$$

We design the language so that, for all these types, *every* computable strategy for the corresponding game are expressible in Lingay. This requirement has led to the discovery of interesting new language primitives, e.g. for coroutining and backtracking.

Proof uses the fact that every type is a computable retract of the universal type `!(int -o int)`.

Limitations of the game model

Our simple game setting can't directly model:

- Name generation, e.g. references with equality
- Cyclic heap structures
- Unrestricted higher order store.

However, encapsulated uses of these features are fine, and are provided for by the following construct:

```
reftype r for T in
   ... val x = ref r e ...
   ... x = x' ...
   ... deref x ...
end
```

This 'controlled' use of higher-order store also leads to better static regulation of exceptions (more flexible than Java, safer than SML).

Implementation

Our game semantics is naturally executable: the compositional definition of $[\![-]\!]$ directly yields an SML function

$$\text{(Lingay syntax trees)} \quad \longrightarrow \quad \text{(strategies)}$$

- Not fast, but good to have a reference implementation.
- Definition uses 'high-level' categorical combinators.
- Can get execution traces for object interactions for free.
- Can interactively play against the strategy for a program.

Further work . . .

- Write some programs! Experiment with programming idioms made possible by new language features.

- Use game semantics for reasoning about programs.
  - Conduct ad hoc proofs.
  - Design program logics inspired by semantics.

- Consolidate metatheoretical results.

- Complete the definition of Eriskay.

Conclusion

Our results so far seem to support the view that . . .

A simple mathematical model
can lead to a
harmonious language design
with some unexpected good properties.

Project website: `http://homepages.inf.ed.ac.uk/jrl/Eriskay`