Eriskay: a programming language based on game semantics

John Longley and Nicholas Wolverson

6 April 2008

School of informatics

Outline

- Motivation
- Model and basic language
- Control features
- Objects and classes
- The *argument-safe* type system
- Conclusions

Motivation

The Eriskay project: Use a simple mathematical model of computation (a game model) to guide the design of a full-scale programming language.

We have in mind a strongly typed, higher order, polymorphic, class-based, object-oriented language, inspired by languages such as Java and ML. Some motivations:

- Reasoning about programs. Logical full abstraction means that logics derived from the model can be understood in terms of the language.
- "Hygiene" properties. Semantically based language design promises to yield properties like type safety and security for exceptions, continuations, name generation.
- Expressive new constructs suggested by model.

OO

Game semantics is intuitively a good match for object-oriented languages:

- Can model stateful computation.
- Good for *data abstraction*. We can interpret an object as a strategy for its externally observable behaviour, and gain a full abstraction result.
- Captures the idea of *reactive* computation (an ongoing interaction rather than a final result)

We consider a *core language* which can interpreted simply in our game model, and a *full language* including more problematic features (references with equality) which require some extra effort to model. (Also cut-down language Lingay)

Introduction to Eriskay

Eriskay is a strongly typed class-based object-oriented language, with

- Objects with mutable state
- Functions (and recursion), sums, (labelled) products
- Recursive types, structural subtyping and System F style polymorphism (and F-bounded)
- Linear type system
- A form of continuations

Game model

We work in the simple category of Lamarche games—games are just trees of alternating Opponent/Player moves, with no restrictions such as well-bracketing. Define games $\otimes$, $\multimap$, etc.

There are two linear exponentials '!' of particular interest:

- Hyland exponential—$!A$ is simply an infinitary (ordered) product of the game A.
- Backtracking exponential—each move in $!A$ may continue play in some copy of A, or backtrack to some move and open a new copy.

Basic language features

Types:

$$\sigma \;::=\; \texttt{int} \mid \sigma_1\texttt{*}\sigma_2 \mid \sigma_1\texttt{+}\sigma_2 \mid \sigma_1\texttt{->}\sigma_2 \mid \texttt{!}\sigma_1 \mid \{l_1\texttt{:}\sigma_1\texttt{,}\ \ldots\ \texttt{,}l_n\texttt{:}\sigma_n\}$$

Language is strict, plain functions are linear and not reusable:

$$[\![\sigma_1\texttt{->}\sigma_2]\!] = [\![\sigma_1]\!] \multimap [\![\sigma_2]\!]_\perp$$

Records are labelled products:

$$[\![l_1\texttt{:}\sigma_1\texttt{,}\ \ldots\ \texttt{,}l_n\texttt{:}\sigma_n]\!] = [\![\sigma_1]\!] \otimes \ldots \otimes [\![\sigma_n]\!]$$

Catchcont

We define a control operator `catchcont` providing a form of resumable exceptions (in various flavours). Where ρ, τ are ground types:

$$\frac{x : \rho\texttt{->}\sigma \;\vdash\; e : \tau}{\begin{array}{l}\vdash \texttt{catchcont}_1\ x \texttt{=>} e \\ \quad : \ \{\texttt{result} : \tau\}\ \texttt{+} \\ \qquad \{\texttt{arg} : \rho,\ \texttt{resume} : \sigma\texttt{->}\tau\}\end{array}}$$

$$\frac{x : \texttt{!}(\rho\texttt{->}\sigma) \;\vdash\; e : \tau}{\begin{array}{l}\vdash \texttt{catchcont}_2\ x \texttt{=>} e \\ \quad : \ \{\texttt{result} : \tau\}\ \texttt{+} \\ \qquad \{\texttt{arg} : \rho,\ \texttt{resume} : \sigma\texttt{->!}(\rho\texttt{->}\sigma)\texttt{->}\tau\}\end{array}}$$

Catchcont, continued

Semantic considerations suggest a more general operator:

$$\frac{x : \texttt{!}(\rho\texttt{->}\sigma) \ \vdash \ e : \tau\texttt{*}\tau'}{\begin{array}{l}\vdash \texttt{catchcont}_3\ x \texttt{=>} e \\ \quad : \ \{\texttt{result}: \tau\texttt{,}\, \texttt{more}: \texttt{!}(\rho\texttt{->}\sigma)\texttt{->}\tau'\} \ \texttt{+} \\ \quad\quad \{\texttt{arg}: \rho\texttt{,}\, \texttt{resume} : \sigma\texttt{->!}(\rho\texttt{->}\sigma)\texttt{->}\tau\texttt{*}\tau'\}\end{array}} \quad \rho, \tau \text{ ground}$$

To show definability and full abstraction we consider the *universal game* $U = [\![\texttt{!(int->int)}]\!]$. All computable strategies of U are language-definable, and basic types, $U \otimes U$, $U \oplus U$, $U \multimap U$, $!U$ and $U_\perp$ are all definable retracts of U.

Coding the retraction $(U \multimap U) \to U \lhd U$ makes use of the power of $\texttt{catchcont}_3$.

Catchcopy

Under the backtracking interpretation of ‘!’, we additionally have a reusable version:

$$\frac{x : \texttt{!}(\rho\texttt{->}\sigma) \;\vdash\; e : \tau\texttt{*}\tau'}{\begin{array}{l}\vdash \texttt{catchcopy}\ x \texttt{=>} e \\ \quad : \ \{\texttt{result}:\tau\texttt{,}\ \texttt{more}:\texttt{!}(\texttt{!}(\rho\texttt{->}\sigma)\texttt{->}\tau')\}\ \texttt{+} \\ \qquad \{\texttt{arg}:\rho\texttt{,}\ \texttt{resume} : \sigma\texttt{!}(\texttt{->}\texttt{!}(\rho\texttt{->}\sigma)\texttt{->}\tau\texttt{*}\tau')\}\end{array}} \quad \rho, \tau \text{ ground}$$

Again, this is required for definability.

Classes

For now assume that methods are *public*, and fields are *protected*. A class implementation is a first-class expression of type `classimpl` τ_f, τ_m, τ_k, where:

- τ_f is a record type for the fields,
- $\tau_m = \{m_1\texttt{:!}(\rho_1\texttt{->}\rho_1'), \ldots, m_n\texttt{:!}(\rho_n\texttt{->}\rho_n')\}$ is the type for objects of the class
- τ_k is the argument type for the (single) constructor

Given such a class implementation c, we can construct an object via the expression `constr` $c\texttt{:}\tau_k\texttt{->}\tau_m$.

But what does one look like?

Method bodies

For object type τ_m, with fields of type τ_f, the method bodies will have type $\tau_m \natural \tau_f$.

In the case of the Hyland !, there is a 'functional' treatment of state:

$$\tau_m \natural \tau_f = \{m_1 \texttt{:!(} \rho_1 \texttt{*} \tau_f \texttt{->} \rho'_1 \texttt{*} \tau_f \texttt{)}, \ldots, m_n \texttt{:!(} \rho_n \texttt{*} \tau_f \texttt{->} \rho'_n \texttt{*} \tau_f \texttt{)}\}$$

With the backtracking !, we can introduce more flexible *read* and *write* operations:

$$\tau_m \natural \tau_f = \texttt{!(!(\{\}->}\tau_f\texttt{)->!(}\tau_f\texttt{->\{\})->}\tau_m\texttt{)}$$

(Note: not every expression of either of these types is a suitable method body)

Class implementations

In a class body, we leave 'open' the method implementations, via a parameter $self : \tau_m \natural \tau_f$, allowing for *method overriding*.

A class is interpreted via the resulting approximation operator $\tau_m \natural \tau_f \texttt{->} \tau_m \natural \tau_f$. The fixed point of this is taken at object creation time.

An additional parameter $super$ can be added, and to allow for additional fields in subclasses we can replace τ_f by $\tau_f \texttt{*} \delta$ (unfortunately not $\alpha <: \tau_f$).

$$\frac{\begin{array}{c} c : \texttt{classimpl}\ \tau_f \texttt{,} \tau_m \texttt{,} \tau_k \\ e_m : \texttt{polytype}\ \delta \texttt{=>}\ \tau_{super} \texttt{->} \tau_{self} \texttt{->} \tau_{self} \\ e_k : \tau'_k \texttt{->} \tau_k \texttt{*} (\tau_f \texttt{->} \tau'_f) \end{array}}{\texttt{extend}\ c\ \texttt{with}\ e_m \texttt{,} e_k : \texttt{classimpl}\ \tau''_f \texttt{,} \tau''_m \texttt{,} \tau'_k}$$

$$\begin{array}{l} \tau_{super} = \tau_m \natural (\tau''_f \texttt{*} \delta) \\ \tau_{self} = \tau'_m \natural (\tau''_f \texttt{*} \delta) \\ \tau''_f = \tau_f \sharp \tau'_f \\ \tau''_m = \tau_m \sharp \tau'_m \\ \tau_f, \tau'_f \text{ have disjoint labels} \end{array}$$

Restrictions on higher-order store

Our class implementations seem to allow us to define a higher-order store cell. Suppose s is a store cell for (`int->int`), and we run

$$s.put(\texttt{fn } x\texttt{=>}x);\ s.get()\ 5$$

We get 'bad' behaviour:

	put:(`int -> int`) `->` {},		get:{} `->` (`int -> int`)	
O		?		
P		!		
O			?	
P			!	
O				?5
P	?5			

Argument safety

Problematic behaviour occurs when a method argument is accessed via the state after the method returns. The type system ensures the property of *argument safety*, that this does not occur.

New judgement forms such as '$\Gamma \vdash e : \tau\ \mathrm{safe}$'.

Fundamental principle: information from an argument may only flow into the state via an expression of ground type.

This means that our language does not permit arbitrary uses of higher-order store; on the other hand, we are not restricted to ground type store.

What do we have

- Can create objects with higher-type fields (f): $\texttt{new}\ C\ (x\texttt{:int->int})$ can set $f := x$
- Cannot store a non-ground-type argument: $m(x\texttt{:int->int})\{f := x\}$.
- Cannot store a non-ground-type value obtained from argument:

$$m(x\texttt{:int->int->int})\{f := x\ 5\}$$

- Can interact with fields: $m()\{\texttt{return}\ (f\ 5)\}$
- Update non-ground fields: $m()\{f := \lambda n.\ f\ n\ +\ 1\}$
- Make use of ground type info from argument $m()\{p := x5; f := \lambda n.\ p\}$
- Use fields and arguments unrestrictedly in return values:

$$m(x\texttt{:int->int})\{\texttt{return}\ (f, x)\}$$

Exception safety

Argument safety has applications to statically controlled exceptions.

- In ML, it is possible for an exception to escape its static scope.

- Conversely, Java's typing of exceptions can be too restrictive.

Consider the Java program:

```
interface Function {Element f (Element x);}
interface List {void add (Element x);
                void map (Function F);
                Element nth (int n);}
```

Intuitively, `map` is argument safe, while `add` is not.

Future work

- Implementation (coming soon)
- Soundness proof (extension of proof for smaller language)
- Details of full language
- Program logics etc.

Conclusions