Eriskay: a programming language based on game semantics

John Longley and Nicholas Wolverson

6 April 2008
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.
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 $\otimes$, $\rightarrow$, 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:

\[ \sigma ::= \text{int} \mid \sigma_1 \sigma_2 \mid \sigma_1 + \sigma_2 \mid \sigma_1 \rightarrow \sigma_2 \mid ! \sigma_1 \mid \{l_1: \sigma_1, \ldots, l_n: \sigma_n\} \]

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

\[ [\sigma_1 \rightarrow \sigma_2] = [\sigma_1] \rightharpoonup [\sigma_2] \perp \]

Records are labelled products:

\[ [l_1: \sigma_1, \ldots, l_n: \sigma_n] = [\sigma_1] \otimes \ldots \otimes [\sigma_n] \]
Catchcont

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

\[
\begin{align*}
  x : \rho \rightarrow \sigma & \vdash e : \tau \\
  \vdash catchcont_1 \ x \Rightarrow e \\
  & : \{ \text{result: } \tau \} + \\
  & \quad \{ \text{arg: } \rho, \text{resume: } \sigma \rightarrow \tau \} \\
\end{align*}
\]

\[
\begin{align*}
  x : ! (\rho \rightarrow \sigma) & \vdash e : \tau \\
  \vdash catchcont_2 \ x \Rightarrow e \\
  & : \{ \text{result: } \tau \} + \\
  & \quad \{ \text{arg: } \rho, \text{resume: } \sigma \rightarrow ! (\rho \rightarrow \sigma) \rightarrow \tau \} \\
\end{align*}
\]
Catchcont, continued

Semantic considerations suggest a more general operator:

\[
\frac{x : !(\rho \rightarrow \sigma) \vdash e : \tau\tau'}{\vdash \text{catchcont}_3 \ x \Rightarrow e}
\]

\[
\rho, \tau \text{ ground}
\]

\[
: \{\text{result: } \tau, \text{ more: } !(\rho \rightarrow \sigma) \rightarrow \tau'\} +
\{\text{arg: } \rho, \text{ resume: } \sigma \rightarrow ! (\rho \rightarrow \sigma) \rightarrow \tau\tau'\}
\]

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

Coding the retraction \((U \rightarrow U) \rightarrow U \sqtriangleleft U \) makes use of the power of catchcont\(_3\).
Catchcopy

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

\[
x : !(\rho \to \sigma) \vdash e : \tau \ast \tau'
\]

\[\vdash \text{catchcopy } x => e\]

\[: \{\text{result: } \tau, \text{more: } !(\!(\rho \to \sigma) \to \tau')\} + \{\text{arg: } \rho, \text{resume: } \sigma!(\rightarrow !((\rho \to \sigma) \to \tau \ast \tau'))\}\]

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 \( \text{classimpl} \ \tau_f, \tau_m, \tau_k \), where:

- \( \tau_f \) is a record type for the fields,
- \( \tau_m = \{ m_1 : ! (\rho_1 \rightarrow \rho'_1), \ldots, m_n : ! (\rho_n \rightarrow \rho'_n) \} \) is the type for objects of the class
- \( \tau_k \) is the argument type for the (single) constructor

Given such a class implementation \( c \), we can construct an object via the expression \( \text{constr } c : \tau_k \rightarrow \tau_m \).

But what does one look like?
Method bodies

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

In the case of the Hyland $!$, there is a ‘functional’ treatment of state:

$$\tau_m \triangleleft \tau_f = \{ m_1 : ! (\rho_1 \ast \tau_f \rightarrow \rho'_1 \ast \tau_f), \ldots, m_n : ! (\rho_n \ast \tau_f \rightarrow \rho'_n \ast \tau_f) \}$$

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

$$\tau_m \triangleleft \tau_f = ! (! (\{\} \rightarrow \tau_f) \rightarrow ! (\tau_f \rightarrow \{\}) \rightarrow \tau_m)$$

(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
\( \text{self} : \tau_m \downarrow \tau_f \), allowing for \textit{method overriding}.

A class is interpreted via the resulting approximation operator \( \tau_m \downarrow \tau_f \rightarrow \tau_m \downarrow \tau_f \). The fixed point of this is taken at object creation time.

An additional parameter \textit{super} can be added, and to allow for additional fields in subclasses we can replace \( \tau_f \) by \( \tau_f \ast \delta \) (unfortunately not \( \alpha \prec \tau_f \)).

\[
\begin{align*}
\text{c: classimpl } & \tau_f, \tau_m, \tau_k \\
\text{e_m: polytype } & \delta \rightarrow \tau_{\text{super}} \rightarrow \tau_{\text{self}} \rightarrow \tau_{\text{self}} \\
\text{e_k: } & \tau_k' \rightarrow \tau_k \ast (\tau_f \rightarrow \tau_f') \\
\text{extend c with e_m, e_k: classimpl } & \tau_f'', \tau_m'', \tau_k'
\end{align*}
\]

\[
\begin{align*}
\tau_{\text{super}} &= \tau_m \downarrow (\tau_f'' \ast \delta) \\
\tau_{\text{self}} &= \tau'_m \downarrow (\tau_f' \ast \delta) \\
\tau_f'' &= \tau_f \# \tau_f' \\
\tau_m'' &= \tau_m \# \tau_m' \\
\tau_f, \tau_f' \text{ have disjoint labels}
\end{align*}
\]
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 $(\text{int} \to \text{int})$, and we run

$$s.put(\text{fn } x \mapsto x); \ s.get() \ 5$$

We get ‘bad’ behaviour:

\[
\begin{array}{c}
\text{put} : (\text{int} \to \text{int}) \to \{\}, \ \text{get} : \{\} \to (\text{int} \to \text{int}) \\
O & ? \\
P & ! \\
O & ? \\
P & ! \\
O & ?5 \\
P & ?5
\end{array}
\]
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 ‘Γ ⊢ e : τ 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} \(C \; (x:\text{int}\rightarrow\text{int})\) can set \(f := x\)
- Cannot store a non-ground-type argument: \(m(x:\text{int}\rightarrow\text{int}) \{ f := x \}\).
- Cannot store a non-ground-type value obtained from argument:
  \[
  m(x:\text{int}\rightarrow\text{int}\rightarrow\text{int}) \{ f := x \, 5 \}
  \]
- Can interact with fields: \(m()\{\text{return} \; (f \, 5)\}\)
- 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:
  \[
  m(x:\text{int}\rightarrow\text{int})\{\text{return} \; (f, \; x)\}\]
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:

```java
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