# Cruise
# A Language of Bad Actors

Mark C. Chu-Carroll

November 10, 2006

**Abstract**

Cruise is a pathological language based on Gul Agha's Actor model. It's a really awful programming language with pretensions of adequacy. And when I think of awful actors with delusions of adequacy, what name comes to mind? Cruise.

## 1 Introduction

This is an introduction to a really crappy programming language. The basic idea of it is to take the Actors model, and build a language around it. So to understand Cruise, you'll need to understand a tiny bit about actors.

### 1.1 The Actor Model

Actors are a theoretical model of computation, which is designed to describe completely asynchronous parallel computation. Doing things totally asynchronously is very strange, and very counter-intuitive, but in real distributed systems, everything *is* fundamentally asynchronous, being able to describe distributed systems in terms of a simple, analyzable model is a good thing. And you *can* build a pretty good programming language based on it; but this isn't it.

According to the actor model, a computation is described by a collection of things called, what else, actors. An actor has a *mailbox*, and a behavior. The mailbox has a unique name, and a queue onto which messages The behavior describes what it's going to do when it processes the next message in its queue. What the actor does is read a message from its queue, and process it according to its current behavior. The behavior gets to look at the message, and based on what the message says, it can do three things:

1. Create other actors.

2. Send messages to other actors.

3. Specify a new behavior for the actor to use to process its next message.

You can do pretty much anything you need to do in computations with that basic mechanism. The catch is, as I said, it's all asynchronous. So, for example, if you want to write an actor that adds two numbers, you *can't* do it by what you'd normally think of as a subroutine call. Subroutines are synchronous; think of it as something like Smalltalk: in Smalltalk, you do things by sending messages to other objects, and when you send a message, you stop and wait until the receiver of the message is done with it. In Actors, it doesn't work that way: you send a message, and it's sent, over and done with. If you want a reply, you need to send the the other actor a reference to your mailbox, and make sure that your behavior knows what to do when the reply comes in.

## 2 The Cruise Language

There are two main things to understand for writing Cruise languages: how to define actors; how to represent data. We'll start by talking about data, which is represented by *tagged tuples*.

### 2.1 Tuple Data

Cruise has a strange data model. The idea behind it is to make it easy to build actor behaviors around the idea of pattern matching. The easiest/stupidest way of doing this is to make all data consist of tagged tuples. A tagged tuple consists of a tag name, and a list of values enclosed in the tuple. The values inside of a tuple can be either other tuples, or actor names.

So, for example, `Foo(Bar(), adder)` is a tuple. The tag is "`Foo`". It's contents are another tuple, "`Bar()`", and an actor name, "`adder`".

Since tuples and actors are the only things that exist, we need to construct all other types of values from some combination of tuples and actors. To do math, we can use tuples to build up Peano numbers. The tuple "`Z()`" is zero; "`I(`*n*`))`" is the number "n+1". So, for example, 3 is "`I(I(I(Z())))`".

The only way to decompose tuples is through pattern matching in messages. In an actor behavior. message handlers specify a *tuple pattern*, which is a tuple where some positions may be filled by*unbound* variables. When a tuple is matched against a pattern, the variables in the pattern are bound to the values of the corresponding elements of the tuple.

A few examples:

- matching `I(I(I(Z())))` with `I($x)` will succeed with $x bound to `I(I(Z))`.
- matching `Cons(X(),Cons(Y(),Cons(Z,Nil())))` with `Cons($x,$y)` will succeed with $x bound to `X()`, and $y bound to `Cons(Y(),Cons(Z(),Nil()))`.

- matching `Cons(X(),Cons(Y(),Cons(Z(),Nil())))` with `Cons($x, Cons(Y(), Cons($y, Nil())))` will succeed with \$x bound to `X()`, and \$y bound to `Z()`.

## 2.2 Defining Actor Types

An actor type is something like a class declaration in a normal language. It defines a type of actor, the messages it can handle, and the behaviors it can follow. The basic pattern of an actor type declaration is:

```
actor !ActorTypeName {
   ( BehaviorDecl )+
   initial :BehaviorName
}
```

A behavior declaration declares a name for the behavior; a possibly empty list of *state variables* that will be assigned values when the behavior is adopted; and a set of message handling clauses:

```
behavior :Name ( CommaSeparatedVariableList ) {
   ( MessageHandler )+
}
```

Each message handlers specify a *pattern*; the handlers are checked in order, and the first one whose pattern matches a message is executed. The body of the handler can create other actors, send messages, and specify a new behavior for the actor to adopt. The adopted behavior must be one of the behaviors defined for the actor type, and it must have a parameter list that matches the parameter list of the behavior declaration:

```
on Pattern {
   (ActorCreateStatement )*
   ( MessageSendStatement )*
   ( adopt :BehaviorName ( CommaSeparatedValueList )
}
```

An actor creation statement in a message handler takes the form `instantiate` *`!ActorTypeName`* `(` *`CommaSeparatedParamList`* `)` `to` *`$VariableName`*. This creates an actor of the specified type, starting its initial behavior with the specified parameters. The name of the actor is assigned to the variable name.

A message send statement is simple: `send` *`TupleValue`* `to` *`ActorName`*. The actor name can be either a literal actor name, or a variable whose value is an actor name.

## 2.3 Main Body

The main body of the program consist of a list of instantiations, followed by a list of message sends. The instantiations here, instead of instantiating to a variable, specify the name of the actor:

`instantiate` *`:ActorType`* `as` *`actorname`*

There should be some examples here.