A Semantics for Exception Handling in Orc

Andrew Matsuoka and David Kitchin

August 14, 2009

This document describes the semantics for the exception handling capabilities of the Orc programming language. It is split into two parts: the first introduces the syntax and behavior of exceptions, and the second outlines an extension of the Orc calculus, called *chromatic* Orc, which can be used to desugar exception handling forms and reason about their behavior. Chromatic Orc also allows a more compact and stateless implementation of certain programming idioms, such as *routing*, that were previously only implementable using channels.

1 Exceptions in Orc

We introduce two new expressions into the Orc programming language:

`throw` E: Execute expression E. For each value v published by E, raise an exception carrying value v. This consumes the publication of v. Since all publications of E are consumed in this way, **`throw`** E is silent.

`try` E **`catch`** `(x)` H: Execute E, allowing it to publish values as normal. Whenever an exception is raised by E, the value carried by that exception is bound to `x` in a new copy of H. E continues to execute as normal, and the copy of H executes in parallel with it.

This treatment of exception handling differs from conventional sequential programming in two important ways. First, **`throw`** may raise multiple exceptions, possibly in parallel. Second, when an exception is caught, its handler executes in parallel with the continued execution of the expression that threw the exception; that is, exceptions do not cause termination.

1.1 throw

Here are some examples of **throw**:

```
throw 3
throw "error"
throw (3 | 4)
throw stop
```

Exceptions are propagated through the other combinators without any effect; they are entirely distinct from publications. In the following two examples, site `M` is never called:

```
throw 3 >x> M(x)
M(y) <y< throw 4
```

1.2 try/catch

Here is a simple example with **try** and **catch**:

```
try
  3 | throw 4
catch (x)
  (1 - x)
```

Patterns are allowed within a **catch**. In fact, multiple **catch** clauses may be listed in sequence, which are each tried in order, just like clausal function definitions. If no clause matches, then the exception is rethrown. Here is an example:

```
try (0 | throw 1 | throw 2)
catch (1) "one"
catch (x) x
catch (2) "two"
```

The following program publishes `0`, `"one"`, and `2`. Notice that `"two"` is not publishes, since the clause **catch** `(x)x` is matched first. Here is an example with a pattern failure:

```
try (
  try (throw 3)
  catch (1) "one"
)
catch (x) x+1
```

The outer handler catches the exception and publishes `4`.

Note that active exception handlers are subject to termination just like any other Orc expression. When a **try/catch** expression is terminated, the body expression terminates, and all running handlers also terminate. Consider this example:

```
stop << (
  Rtimer(2000)
  | (
      try (
        Rtimer(1000) >> throw 0
        Rtimer(3000) >> throw 1
      )
      catch (x)
        M(x) >> N(x) >> P(x) >> Q(x)
    )
)
```

An exception with value `0` is thrown and caught after 1000 ms, causing four site calls to execute in series. At 2000 ms, the entire expression is terminated, including the running handler, which may not have finished calling all of the sites. The expression **throw** `1` will never even be executed.

1.3 Exceptions and halting

The **throw** and **try/catch** expressions halt as one might expect: !throw E! halts when E does, and **try** E **catch** (p) H halts when E has halted and all active handlers H have also halted. Thus, these expressions may be combined with the otherwise combinator; for example, **throw** `0` **;** `1` will throw an exception and subsequently publish `1`.

1.4 Exceptions from sites

A site call may also throw an exception, instead of publishing a value; if it does so, it then halts immediately. In the current Java-based implementation, sites implemented in Java may even throw exceptions within Java code, which are then propagated into the Orc program, where they may be caught by an Orc exception handler.

2 Chromatic Orc

One of the fundamental principles in the design of the Orc programming language is that each new construct should be representable in terms of the underlying calculus. However, exceptions are very difficult to encode in the current Orc calculus; such an encoding would require a burdensome, global program transformation.

So instead, we present a slightly extended version of the Orc calculus, into which exception handling can be encoded very directly. We do this by giving *colors* to publications and to combinators; for example, by giving normal publications one color, and exceptional publications a different color, we can allow exceptions to propagate through normal combinator uses.

2.1 Adding Color

Site return events and publications now have an associated *color* c; they are written $k?_c v$ and $!_c v$ respectively. The site *let* becomes a family of sites indexed by colors c; the site call $let_c(v)$ publishes value v with color c.

We remove the $|$ combinator, and add a subscript s, called a *spectrum*, to each of the other combinators: $f >x>_s g$, $f <x<_s g$, $f ;_s g$. A spectrum is either a color c, or the special marker $\bullet$, which indicates an absence of color. A publication with color c is handled by a combinator with spectrum c in the same way as in the previous Orc semantics. Publications with colors other than c are allowed through, just as if they were non-publication events. The marker $\bullet$ is not equal to any color; a combinator with spectrum $\bullet$ allows all publications through.

The expressions $f \gg_s g$ and $f \ll_s g$ are shorthand for $f >x>_s g$ where g is x-free, and $f <x<_s g$ where f is x-free, respectively.

2.2 Embedding the Orc calculus

The conventional Orc calculus, plus exception handling, can be encoded in chromatic Orc as follows. Distinguish two colors: p (for publications) and e (for exceptions).

$$
\begin{array}{rcl}
f >x> g & = & f >x>_p g \\
f <x< g & = & f <x<_p g \\
f \,;\, g & = & f ;_p g \\
f \mid g & = & f \ll_\bullet g \\
let(x) & = & let_p(x) \\
\texttt{throw}\ f & = & f >x>_p let_e(x) \\
\texttt{try}\ f\ \texttt{catch}\ (x)\ g & = & f >x>_e g
\end{array}
$$

Sites are allowed to choose the color of their return values. Typically the chosen color will be p, but a site might color its return value as e to throw an exception directly.

2.3 Alternate semantics for exceptions

Suppose that we chose to use $\ll$ instead of $\gg$ in the encoding of try/catch. Then the semantics of exception handling would be closer to the understanding of exceptions in sequential programming: terminate the expression which raised the exception, and subsequently execute a handler (or rethrow, if no handler matches). Since the particular expression from which the exception originated always halts, the sequential model is simply a conflation of both of these options: when there is only one thread of control, it is impossible to distinguish terminating that thread of control from terminating all threads of control.

In Orc, we have chosen the $\gg$ interpretation, since it more clearly highlights the options available in concurrent exception handling, and because it is much easier to provoke termination (perhaps by putting a $\ll$ just outside the scope of the try/catch) than to prevent termination.

2.4 Alternate semantics for ;

Early in the design of the ; combinator, we considered two possible interpretations. The choice represented by $f ;_p g$ is now the canonical one: if f halts without having published any values, execute g. The other option, represented by $f ;_\bullet g$, was: run f until it halts and then run g regardless of whether f published or not. This interpretation is more useful in certain contexts and may be more intuitive to users of sequential programming languages. The chromatic combinators allow both interpretations to coexist with a unified semantics.

2.5 Operational Semantics

The operational semantics of chromatic Orc is given in Figure 1. It is quite similar to the semantics of the non-chromatic calculus.

2.6 Algebraic Laws

(Left zero of $\gg$)	$\mathtt{stop} >x>_s f = \mathtt{stop}$
(Left unit of $\gg$)	$\mathtt{signal}_c \gg_{\mathbf{c}} f = f$
(Right unit of $\gg$)	$f >x>_c let_c(x) = f$
(Left unit of $\ll$)	$\mathtt{stop} \ll_\bullet f = f$
(Right unit of $\ll$)	$f \ll_s \mathtt{stop} = f$
(Commutativity of $\ll$)	$(f <x<_s g) <y<_{s'} h = (f <y<_{s'} h) <x<_s g$, if h is x-free and g is y-free
(Associativity of $\gg$)	$(f >x>_s g) >y>_{s'} h = f >x>_s (g >y>_{s'} h)$, if h is x-free
(Commutativity of $\ll$ with $\gg$)	$(f <x<_c g) >y>_c h = (f >y>_c h) <x<_c g$, if h is x-free
(Distributivity of $\ll$ over $\gg$)	$(f <x<_s g) >y>_c h = (f >y>_c h) <x<_s (g >y>_c h)$, if h is x-free and $s \neq c$

There are no rules involving $|$, since it is now encoded via $\ll_\bullet$. All of the $|$ laws are provable from these laws.

2.7 Routing with Chromatic Combinators

Some of the examples of *routing* in the user guide can be rewritten directly using chromatic combinators, rather than channels.

However, not all of the examples can be recast. Publication Limit requires explicit counting, which the chromatic combinators cannot accomplish. Non-Terminating Pruning does not have a clear analogue, since it actually suppresses the terminating behavior of $\ll$ while preserving its binding behavior, rather than just redirecting publications.

2.7.1 Enhanced Timeout

Execute f, allowing it to publish any number of values, until time limit t is reached. Assume that f does not publish values of the special color z.

$$\texttt{stop} \ll_z (f \mid \mathit{Rtimer}(t) \gg \texttt{signal}_z)$$

2.7.2 Interrupt

Similar to enhanced timeout, except that we are waiting for some other party to release the semaphore *done*, rather than waiting for a timeout.

$$\texttt{stop} \ll_z (f \mid \mathit{done.acquire}() \gg \texttt{signal}_z)$$

2.7.3 Test Pruning

Execute f until it publishes a negative number of color p, and then terminate it.

$$x <x<_z (f >x>_p (\texttt{if } (x < 0) \texttt{ then } let_z(x) \texttt{ else } let_p(x)))$$

$$\frac{k \text{ fresh}}{v(\bar{v}) \overset{v_k(\bar{v})}{\rightarrow} ?k}\ (\textsc{SiteCall})$$

$$?k \overset{k?_c v}{\rightarrow} let_c(v)\ (\textsc{SiteRet})$$

$$?k \overset{k?\perp}{\rightarrow} \mathbf{stop}\ (\textsc{SiteHalt})$$

$$let_c(v) \overset{!_c v}{\rightarrow} \mathbf{stop}\qquad (\textsc{Let})$$

$$\frac{(E(x) \triangleq f) \in D}{E(p) \overset{\tau}{\rightarrow} [p/x]f}\ (\textsc{Def})$$

$$\frac{}{\mathbf{stop}\ ;_s\ g \overset{\tau}{\rightarrow} g}\ (\textsc{SemiZ})$$

$$\frac{f \overset{l}{\rightarrow} f' \qquad l \text{ is not a } !}{f\ ;_s\ g \overset{l}{\rightarrow} f'\ ;_s\ g}\ (\textsc{SemiN})$$

$$\frac{f \overset{!_c v}{\rightarrow} f' \qquad c = s}{f\ ;_s\ g \overset{\tau}{\rightarrow} f'}\ (\textsc{SemiC})$$

$$\frac{f \overset{!_c v}{\rightarrow} f' \qquad c \neq s}{f\ ;_s\ g \overset{!_c v}{\rightarrow} f'\ ;_s\ g}\ (\textsc{SemiNC})$$

$$\frac{f \overset{l}{\rightarrow} f' \qquad l \text{ is not a } !}{f >x>_s g \overset{l}{\rightarrow} f' >x>_s g}\ (\textsc{SeqN})$$

$$\frac{f \overset{!_c v}{\rightarrow} f' \qquad c = s}{f >x>_s g \overset{\tau}{\rightarrow} (f' >x>_s g) \mid [v/x]g}\ (\textsc{SeqC})$$

$$\frac{f \overset{!_c v}{\rightarrow} f' \qquad c \neq s}{f >x>_s g \overset{!_c v}{\rightarrow} f' >x>_s g}\ (\textsc{SeqNC})$$

$$\frac{f \overset{l}{\rightarrow} f'}{f <x<_s g \overset{l}{\rightarrow} f' <x<_s g}\ (\textsc{PruneL})$$

$$\frac{g \overset{l}{\rightarrow} g' \qquad l \text{ is not a } !}{f <x<_s g \overset{l}{\rightarrow} f <x<_s g'}\ (\textsc{PruneN})$$

$$\frac{g \overset{!_c v}{\rightarrow} g' \qquad c = s}{f <x<_s g \overset{\tau}{\rightarrow} [v/x]f}\ (\textsc{PruneC})$$

$$\frac{g \overset{!_c v}{\rightarrow} g' \qquad c \neq s}{f <x<_s g \overset{!_c v}{\rightarrow} f <x<_s g'}\ (\textsc{PruneNC})$$

Figure 1: Operational Semantics of Chromatic Orc