1 Orc

Orc is all about “orchestration”. It’s basically the communication, distribution skeleton of a program. Web scripting, workflow applications etc. Orc is the glue for building these orchestration tasks. Orc’s key abstraction is a site, something you can call and that publishes results. Services are implemented as sites.

Orc is a simple language. It consists of

- Site calls $M(v)$
- Symmetric parallel composition $f | g$
- Sequential composition with respect to a variable $x$: $f > x > g$. $f$ executes after $g$ publishes a value, with $x$ bound to that value. If $g$ publishes more than one value, $f$ is run multiple times, once for each value published.
- Asymmetric parallel composition with respect to $x$: $f < x < g$ Subcomputations of $f$ that don’t depend upon $x$ execute in parallel with $g$, while computations dependent upon $x$ block until $g$ publishes a value which is in turn bound to $x$. $f$ is run at most once.
- Definitions $D(x) =_{df} g$

**Site calls** Site calls perform a computation and publish at most one result.

**Symmetric parallel composition** To evaluate $f | g$, evaluate $f$ and $g$ in parallel. $f | g |$ publishes $v$ iff $f$ or $g$ publishes $v$.

**Sequential composition** To evaluate $f > x > g$, begin by evaluating $f$. For each $v$ published by $f$, evaluate $[v/x]g$ in parallel. $f > x > g$ publishes $w$ iff some $[v/x]g$ publishes $w$.

**Asymmetric parallel composition** To evaluate $f < x < g$, evaluate $f$ and $g$ in parallel. $f$ may block waiting for data from $g(x)$. If $g$ publishes $v$, kill $g$ and continue evaluating $[v/x]f$.

**Question 1.** Is $< x <$ necessary? Can it be encoded using $|$ and $> x >$?

**Comment 1.** From Owen: Seems like it was inspired by Bash, I would really like to use a language like this to write in a command line script.
1.1 Examples

- fork – join = (let(x, y) < x < M) < y < N
- sync = fork – join > x > (f|g)
- delay = (Rtimer(1) >> let(x)) < x < M
- priority = let(x) < x < (N|delay)

2 Timed Trace semantics

Originally, Orc was given an asynchronous, then a “synchronous-but-untimed” semantics. Here we will use a “relative-time” semantics which describe delays from site calls.

\[(Rtimer(s) >> let(v)) | (Rtimer(3) >> let(w))\]

The operational semantics are based on a labelled transition system \( f \xrightarrow{t,a} f' \) with time-event pairs \( t, a \) for labels.

Expression \( f \) may engage in event \( a \) after \( t \) units of time, without engaging in other events, resulting in expression \( f' \).

2.1 Rules

Sites

\[\text{let}(v) \xrightarrow{0,v} 0\]

Immediately publish value \( v \) and transition to an expression 0 that engages in no other events.

\[Rtimer(t) \xrightarrow{t,} 0\]

Publish a signal after \( t \) time units.

Combinators

\[
\begin{align*}
&f \xrightarrow{t,a} f' \\
&f|g \xrightarrow{t,a} f'|g
\end{align*}
\]

Works in asynchronous system, but NOT with time. Consider \( Rtimer(8)|Rtimer(3) \).

To fix this, we introduce “Time Shifting”, \( f^t \). Evaluate for \( t \) time units without an event. For example,

\( Rtimer(5)^5 \equiv Rtimer(2) \). But, it may not always be possible: \( Rtimer(5)^7 \not\equiv \perp \).
- \( Rtimer(2)^5 \equiv \perp \)

\[
\begin{align*}
&f \xrightarrow{t,a} f' \\
&f|g \xrightarrow{t,a} f'|g
\end{align*}
\]

Only if \( g^t \) is not \( \perp \).

Rest of the semantics similarly extend the asynchronous semantics.
2.2 Denotational Semantics

An execution is a finite sequence of time-event pairs that \( f \) engages in. A trace is an execution without internal events. \(< f > = \) traces of \( f \) defined operationally.

Trace sets form a denotation, \( \mu(f) \).

**Theorem 1.** Operational and denotation semantics are equivalent: \(< f > = \mu(f)\).

This allows for compositional reasoning.