Continuations and Virtualised Threads

Olin Shivers
MIT AI Lab
shivers@ai.mit.edu

It is well known that concurrency can be expressed within languages that provide a continuation type. However, a number of misconceptions persist regarding the relationship between threads and continuations. In the advanced programming language community, it is common to hear that "a thread is just a continuation." However, it would be more accurate to regard a thread as a *trajectory in continuation space,* (or, equivalently, to regard a continuation as a snapshot of a thread at some point in time). If we take the view that a thread is an abstraction of a processor, then we can view a continuation as thread state, or *an abstraction of processor context.*

This perspective has led to the design of the concurrency primitives in the Express operating system under development at MIT in Standard ML. We expose a low-level CPS form in our programming language for writing low-level concurrent operations. Representing thread state with a continuation allows the compiler to reason about and optimise uses of this state with standard λ-calculus technology. Besides threads and continuations, we also introduce data structures to represent events. An event datum is essentially an *observational capability* on the action of some independent computational agent.

We introduce three operators:

$$
\begin{aligned}
&\texttt{event-dispatch} : (\forall\alpha.\alpha\ \mathit{event} \times \alpha\ \mathit{cont})^* \longrightarrow \\
&\texttt{fork-thread} : \mathit{cont} \longrightarrow \mathit{thread} \\
&\texttt{advance-thread} : \mathit{thread} \times (\forall\alpha.\alpha\ \mathit{event} \times \alpha\ \mathit{cont})^* \longrightarrow
\end{aligned}
$$

The dangling right arrows in the type of `event-dispatch` and `advance-thread` are intended to informally reflect the fact that neither operator returns to its caller—instead, each one invokes a continuation passed as an argument.

The `event-dispatch` operator is the primitive synchronisation operator. It takes an arbitrary number of arguments, each one an event/continuation pair, and blocks, waiting for one of the specified events to occur. When this happens, the thread that called `event-dispatch` resumes with the event's corresponding continuation. In our virtual/physical analogy, where threads, continuations, and events correspond to processors, processor context, and asynchronous external interrupts, the `event-dispatch` operator corresponds to a processor's `halt` instruction, halting a thread until signaled by an external event. The continuations are analogous to the processor's interrupt vector.

The `fork-thread` operator simply creates a thread or virtual processor whose initial context is given by the continuation argument. Once created, the virtual processor must be mapped onto a physical processor in order to progress forwards with its computation. This is accomplished with the `advance-thread` operator.

When a thread t applies `advance-thread` to some sub-thread t', thread t gives all of its processing cycles to t'. Whenever thread t gets cycles, those cycles will actually be passed to t'. This state will persist until one of the events in the *preemption set* occurs—that is, one of the events in the $(\forall\alpha.\alpha\ \mathit{event} \times \alpha\ \mathit{cont})^*$ set of event/continuation pairs passed to `advance-thread`. When such an event happens, t ceases to donate its cycles to t' and resumes computation with the event's corresponding continuation.

If we make sure that "n cpu-seconds from now" and "the next time thread t' blocks" are events in our basic event vocabulary, it is straightforward for a thread to use `advance-thread` to parcel out its cycles to a set of children in a preemptive, non-cooperative manner. It simply loops over the child threads, using `advance-thread` to give a quanta of cycles to each child.

The `advance-thread` operator allows us to implement the illusion of an infinite number of virtual processors on top of a finite number of processors, or even a uniprocessor. A program written using this infinite-processor synchronous thread model will run unchanged on systems with varying numbers of actual processors.

Further, it allows us to completely virtualise our thread model. When a program uses `advance-thread` to pass its thread's cycles to some sub-thread, it is completely unable to detect if its thread is some virtual thread, or an actual hardware processor. A scheduler written to schedule a thread's cycles at user level is exactly the kind of program the OS could use to allocate the use of the actual hardware processors. The model recurses down as far as we like.

Because it encodes pure mechanism, and no policy, the `advance-thread` operator allows users to write a wide range of schedulers for their programs. Note that multiple instantiations of these schedulers can run on multiple threads given to a program by the operating system. These schedulers can use shared data-structures to cooperatively share and schedule a common pool of child threads.