Continuations and threads: Expressing machine concurrency directly in advanced languages

Olin Shivers
MIT AI Lab
Cambridge, Massachusetts, USA
shivers@ai.mit.edu

Abstract

It is well known [Wand] 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. I discuss the proper relationship between these two objects, and present a model for directly expressing concurrency using continuations. The model is designed to support systems programming, and has several novel features: it is synchronous, preemptable, and fully virtualisable, allowing schedulers to be written by unprivileged users that are indistinguishable from top-level schedulers that actually control access to the hardware resources.

1 Introduction

An operating system exports to its client programs protected interfaces to the machine resources, such as memory, persistent storage, network connections, and processors. An important part of the OS design is the model of these resources that the OS presents to the client program. For example, one of the benefits of the Unix operating system is that its "everything is a file" design presents a simple and consistent interface to a wide array of I/O resources

In *Proceedings of the Second ACM SIGPLAN Workshop on Continuations,* January 1997, Paris. Also available as BRICS Notes Series NS-96-13, University of Aarhus, Denmark.

In a similar fashion, one may consider in what fashion an operating system should model the processors it manages—that is, how should the OS export access to the processor? Typical operating systems, such as Unix and Microsoft's NT, use *processes* and *threads* to model computational agents.The MIT Express project is concerned with the design of operating systems for and in advanced, functional programming languages. We have designed a new model for allowing programmers to implement computational agents that execute concurrently on the machine resources that directly models the processor in the programming language. We exploit continuations to do so.

2 Multi-continuations

A linguistic feature we intend to exploit is the ability to define procedures that take multiple continuations as arguments {Note Tops-20}. This will require allowing the programmer to drop down into a CPS sublanguage when it necessary to define or use such procedures. In other words, if the compiler is a CPS-style compiler, then its CPS-converter front-end would pass fragments of the program written in the CPS sublanguage directly through with no further conversion

I do not expect that much CPS code will be written by programmers; it is a low-level form whose use will be limited to definitions of concurrency primitives and frequently hidden behind convenient forms of syntactic sugar by macros. However, it is important that we have this mechanism present for expressing control operators.

For example, the `call-with-current-continuation` or `call/cc` operator can be directly defined in this CPS sublanguage {Note Value/Cont Distinctions}:

```
(call/cc f k) ≡ (f (λ (v k') (k v)) k)
```

We can also push a control frame onto a captured continuation with

```
(compose-cont k f) ≡ (λ (v) (f v k))
```

As we will see later, multi-continuations are useful for expressing concurrency and choices.

3 Synchronous threads

There are several goals our concurrency mechanism needs to satisfy. It should

- be a powerful, expressive model;
- efficiently export the resources of the underlying hardware;
- fit with the language in a harmonious fashion;
- admit optimisation at the language level by standard optimising compiler technology for lambda-calculus-based languages.

The model is based upon concurrent serial computations—individual concurrent computational agents are synchronous processes. An individual computation can be thought of as

- A thread
- A virtual processor
- A computational agent
- A trajectory in continuation space

An important element of this model is that these various viewpoints are equivalent and interchangeable.

4 Problems with interrupt handlers

One striking feature of our model is that it is completely synchronous, by which I mean that asynchronous interrupts are not a part of the model. As an agent evolves over time, its next step is always determined by the program it is executing (which may direct the thread to interact with other, concurrently executing agents, or attend to external events, of course).

Interrupts were ruled out because they have several problems. First, they confound several distinct computational elements, *e.g.*, concurrency, mutex, and communication. It is better to factor these elements into their individual components, allowing the programmer to use only those which are needed by a given application.

Second, the interrupt model doesn't parallelise—it is fundamentally a uniprocessor mechanism. Because the main process is suspended while an interrupt handler runs, there is a mutex "built in" to the handler. This means that if the program is executed on a parallel processor, the system can't run the handler in parallel with the main program—to do so might violate some assumption of mutex made by the programmer.

Third, interrupt handlers are not modular. If one module of code installs a handler for an interrupt, it will clobber the handler installed by any other module. Global resources are always a potential source of conflict.

Finally, there is the argument of economy of mechanism: as we'll see, the thread model functionally dominates the interrupt model, so it is superfluous to provide both mechanisms.

In our synchronous-thread model, if a program wishes to attend to some set of external events, it must synchronously block and wait for an occurrence of the events. If it wishes to compute concurrently while attending to the external events, it must spawn a sub-thread to wait for the event while it performs its main computation.

5 Threads and continuations

Although it is widely known that we can model threads with continuations in programming languages that provide them as first-class values, there are some misunderstandings concerning relationship between these two types.

Myth: thread $=$ continuation

It is common for members of the Scheme community, for example, to declare that *"threads are just continuations."* This belief derives from the fact that, given a means of accessing implicit continuations (such as Scheme's `call/cc` procedure), a simple co-routining thread scheduler can be written in about seven lines of code.

Better: thread $=$ trajectory in continuation space

However, threads are not continuations. It would be better to state that *"A thread is a trajectory in continuation space."*

The standard definition of a continuation is "the rest of the computation," *i.e.*, what remains to be computed after performing some action. As a thread evolves over time, this must change, of course. If a thread has some schedule of actions it must perform, $a_1, a_2, \ldots, a_n$, then after performing actions a_1 through a_4, its remaining schedule is simply $a_5, \ldots, a_n$. Its continuation has changed because it has carried out part of its original continuation.

At each point in time, as the thread performs an atomic action, the thread evolves from one continuation to another. Hence we may view a thread as a trajectory in continuation space, or equivalently, we may view a continuation as a snapshot of a thread at some particular instant.

This observation calls attention to a fundamental distinction between threads and continuations, that of mutability. A continuation, being a description of a thread at some instant, is a functional, immutable value like an integer or a character. In contrast, a thread is an object which has an identity as it evolves over time—an imperative, mutable value. The thread structure performs the function of providing a name for the computation as it sequences through continuation space—a name we can use to reference the computation as we schedule it or otherwise operate upon it over time.

Better: thread = virtual processor

We can also view threads as virtual processors. A thread is an agent of computation that iteratively takes its continuation, and depending upon the value of this continuation, performs some atomic unit of computation, resulting in a new continuation. This is precisely the model of a state machine, where the thread is the machine.

Myth: continuation = stack

A second misconception of continuations is that *"a continuation is just a stack."* This belief derives from the fact that continuations created by operators such as Scheme's `call/cc` are typically created by copying the run-time stack to heap storage.

Continuations created by `call/cc` are special because they are created synchronously: the program explicitly requests the continuation to be constructed at some statically known point in its processing. Because the compiler knows where each `call/cc` occurs, it can emit code to save the live registers and the program counter on the stack, ensure that other processor state such as condition codes is unused, and otherwise "narrow" the processor state down until it can be described by a single value: the stack pointer.

If we step outside the limits of synchronous continuations constructed at points clearly marked to the compiler, we see that a continuation must include the entire processor state. A continuation for a computation that has been preempted or otherwise asynchronously interrupted must include the state of all of the processor registers that might have bearing on the evolution of that computation.

Better: continuation = abstraction of processor state

This leads us to regard a continuation as an abstraction of processor state. The abstraction is exact, in that the state captured by the continuations we find in Scheme or ML includes the processor state *and no more.* Continuations explicitly do not capture the store or the state of the I/O systems. They capture only the state that describes the processor.

This abstraction seems a particularly elegant one. A typical thread package for Unix or Win32 will describe a thread context with some complex C struct [C-threads]. Using continuations instead means that we can expose descriptions of processor state to direct manipulation and optimisation at the language level by standard λ-calculus compiler technology.

6 Real/virtual equivalences

These observations lead us to a fairly strong relation between real, physical processors and synchronous threads:

Concrete	Abstract
Processor	Thread
Register set	Continuation
Interrupt	Event

For example, throwing to a continuation becomes the thread equivalent of a processor “load context” instruction.

Note, however, one major distinction between our threads and real processors: threads are simple, synchronous machines. They do not have interrupt handlers, and do not perceive preemption. Real hardware is interruptible. In this respect, our virtual structures simplify reality.

7 Thread operations

We can use our physical/virtual correspondence to guide the design of our thread system. For example, the mechanism for attending to multiple events is the operator `event-dispatch`.

$$\texttt{event-dispatch} : (\exists\alpha.\alpha\ \mathit{event} \times \alpha\ \mathit{cont})^* \longrightarrow \mathit{Ans}$$

Note that this operator is a multi-continuation operator, taking an arbitrary number of event/continuation pairs. For this reason, we define the operator in a CPS-style declaration—as it does not return a value, we specify some fictitious *Ans* type as its range to represent the eventual completion of the program. In order to use this operator, we need to drop down to an explicit CPS language to pass the multiple continuations it requires (although most uses of this operator would hide it behind some form of branching construct implemented as a macro in terms of `event-dispatch`).

When called, the `event-dispatch` operator blocks the thread and waits for one of the indicated events to occur. When one does, the operator returns through the event's corresponding continuation.

One way to view this operator is that, while a thread can normally be described at some point in time by a single continuation, when the thread blocks in an `event-dispatch` application, it enters a “mixed state,” with multiple possible continuations eligible, each guarded by some associated event. When an external event is observed, the thread's mixed state “snaps” or resolves itself into a single continuation, and the thread can then evolve forward.

This event model is similar to the one employed by CML [CML], but it exposes what in CML is the underlying "canonical event" structure. This means that when the programmer wishes to discriminate between two different events, instead of encoding the difference by tagging the events with `wrap` operators, and then discriminating on the tags, he can directly encode the discrimination by associating two different continuations with the two different events. In other words, we can encode the discrimination "in the pc." Eliminating the encode/decode pair of operations is, of course, more efficient {Note Value/Cont Duality}. Also, continuations are fair game for optimisation by a λ-calculus, CPS-based compiler, so uses of the `event-dispatch` operator are fully exposed to the language-level optimisation machinery.

It is interesting to consider the hardware analog of the `event-dispatch` operator: it is simply the `halt` instruction. A processor's `halt` instruction halts the processor. No matter how often we toggle the processor's clock, it will remain stably in the halt state until an external agent "pokes" the processor by toggling one of its interrupt lines. When this happens, the processor leaves its halt state, and vectors through its interrupt table, resuming its processing at an address that is determined by which of the interrupt lines was toggled.

The `event-dispatch` operator halts its thread, which remains halted until it is activated by the occurrence of an external event. When this happens, the thread leaves its halt state, and resumes its processing at a continuation that is determined by which of the events caused it to re-awaken.

8 Pre-emption and virtualisation

Our synchronous, interrupt-free thread model seems a pleasantly simple one. However, we would also prefer our model be powerful enough to express thread pre-emption. Without pre-emption, it's trivial to write a coroutining thread scheduler using `call/cc`, but we need something more powerful to implement a fair, non-cooperative thread scheduler.

It turns out that by introducing two more operators {Note Thread Inspection}, we can fully virtualise the model, allowing user code to create new virtual processors, and allocate its cycles to sub-threads in a pre-emptive manner. Surprisingly, we will not need to introduce interrupts to achieve this.

The first of our two operators, `fork-thread`, creates new threads:

`fork-thread` : *cont* $\longrightarrow$ *thread*

allocates a virtual processor, and initialises it with some continuation.

The `fork-thread` operator allocates and initialises a virtual processor, but it does not provide a means for "powering" the processor. So the resulting processor does not run forward after being created. For that to happen, it must be given some cycles—it must be connected to a real, physical processor. We can use the second of our two operators, `advance-thread`, to run this virtual thread forwards:

$$\texttt{advance-thread} : \textit{thread} \times (\exists \alpha . \alpha\ \textit{event} \times \alpha\ \textit{cont})^* \longrightarrow \textit{Ans}$$

Note that `advance-thread`, like `event-dispatch`, is a multi-continuation operator, meaning that it can return multiple ways.

When a thread t applies the `advance-thread` operator 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 $(\exists \alpha . \alpha\ \textit{event} \times \alpha\ \textit{cont})^*$ 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. 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 {Note Turtles}.

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.

9 Related work

There is a wealth of related, prior research in the area of continuations and concurrency. The standard work on concurrency models using serial processes with synchronous communications is Hoare's [CSP].

Wand's early paper [Wand] showed means of modeling OS concurrency in Scheme. However, Wand's model was intended more as an explicative aid than a practical tool for use in an actual operating system.

Dybvig has reported on "nested engines," a similar concept for providing pre-emptable schedulers in Scheme [Engines]. A recent release of Scheme 48 [S48] uses a variant of these engines for its thread package.

Reppy's doctoral work on Concurrent ML [CML] provided a concurrency mechanism based on synchronous sequential processes, although Reppy did not consider problems of virtualisation. Reppy has also reported on the nature of continuations in a system with asynchronous interrupts [AsyncCont].

Philbin's doctoral work [Philbin, STNG] on the STNG operating system concerned the design of an operating system based on Scheme. The STNG system provided explicit models of virtual processors and threads in Scheme.

10 Conclusion

The concurrency model we've just examined is neither completely developed, nor is it implemented and tested by experience. Much work remains. Nonetheless, it is a promising model, and leaves us with several observations:

- Explicit CPS is useful for expressing thread-level concurrency in a practical systems-programming environment.
- Representing thread state with continuations can allow compilers to analyse and optimise this state using standard CPS λ-calculus technology.
- Our model is simple (*i.e.*, fully synchronous and interrupt-free), yet general (it still allows for pre-emptive scheduling on finite resources). By carefully choosing our operators, we can not only provide for general scheduling, we can make our processor model fully virtualisable. This is novel and promising.
- Our thread model has close hardware analogs that guide our design of the virtual threads.

My research group at MIT intends to complete the details of this model and implement it as the basic concurrency model for the operating system being designed by the Express project. We hope to report on our results in future publications.

Notes

{CPS/Direct Distinctions}

In these examples, I am deliberately glossing over distinctions between direct-style calls λ-expressions (in which the continuation is implicit), and CPS calls and λ-expressions (in which the the continuations are explicit). A proper CPS sublanguage would distinguish these. For example, if we wrote CPS procedures using μ, and CPS calls using brackets, we would more carefully define our examples as

```
call/cc ≡ (μ (f k) [f (μ (v k') [k v]) k])
compose-cont ≡ (λ (k f) (μ (v) [f v k]))
```

These details, along with issues of continuation typing and the tupling implicit in the s-expression syntax I have employed, are beyond the scope of this note.

{Thread Inspection}

It isn't clear whether we wish to provide thread-inspection and thread-mutation operators. In general, one operates upon a thread by causing events that it can observe, and by scheduling a virtual thread to run pre-emptively with one's own thread. However we could provide other, more invasive operators on our virtual processors.

For example, the administrative operator

`swap-thread-cont!` : *thread* $\times$ *cont* $\longrightarrow$ *cont*

allows us to asynchronously set a thread's continuation to a specific value. Invoking (`swap-thread-cont!` t k) installs k as t's continuation, returning t's continuation at the time of the installation. We can use this operator to kill a thread:

```
(define (kill-thread t)
  (swap-thread-cont! t (λ () (event-dispatch {})))
  #t)
```

or to swap our computation with some other thread's:

```
(define (brain-swap t)
  (call/cc (λ (k) ((swap-thread-cont! t k)))))
```

{**Tops-20**}

There are no new ideas. The Tops-20 operating system employed multiple continuations over twenty years ago [Jsys]. Tops-20 system calls were invoked with the `jsys` instruction, which pushed the current program-counter onto the stack and trapped into the kernel. If the system call completed successfully, the operating system returned to the user program at word pc+2, that is, skipping the word following the `jsys` call. If the system call caused an error, the OS returned to the user program at pc+1. Typical useage was to place a branch to error-handling code immediately after the `jsys` instruction. The rest of the program handling the normal case would simply follow after this branch.

In short, the programmer passed two continuations to the system call: a success and fail continuation, albeit in assembler rather than a high-level language.

{**Turtles**}

An eminent astrophysicist is delivering a public lecture on cosmology. After his lecture has concluded, a slightly eccentric, elderly lady approaches the professor, and says, "Oh, Professor, I enjoyed your lecture very much. But I'm afraid you are entirely mistaken about the structure of the universe and the shape of our world. You see, the earth is actually a large, flat disk sitting on the back of four gigantic elephants."

The professor indulgently smiles, and inquires, "But on what do the elephants stand?"

The little old lady earnestly replies, "Why, they stand on the back of an enormous turtle."

The professor smiles in quiet triumph as he administers his *coup de grace:* "And on what does the turtle stand?"

The little old lady is completely unfazed. Her eyes twinkle, she smiles, and replies, "You can't fool me, young man. Why, after that, it's turtles all the way down."

The application to formal semantics and systems architecture should be clear.

{**Value/Cont Duality**}

We are simply exploiting the equivalence between a continuation that accepts an $A + B$ sum type, and a product of an A continuation and a B continuation. These relations are detailed further in Filinski's master's thesis [Filinski].

Because our multi-continuation CPS form permits us to choose the latter representation, we can encode the discrimination directly in the pc for greater efficiency. In making this design decision, I am indebted to Jonathan Rees for the guidance of his “don't encode” functional programming mantra, which may be expressed as “Don't pass flags encoding actions to procedures which must then discriminate them; instead, pass the actual action to be performed.”

References

[AsyncCont] John H. Reppy. Asnchronous signals in Standard ML. Technical report TR 90-1144, Computer Science Department, Cornell University, August 1990.

[C-threads] Eric C. Cooper and Richard P. Draves. "C threads." CMU Technical Report CMU-CS-88-154. September 1990.

[CML] John H. Reppy. *Higher Order Concurrency.* PhD Dissertation, Cornell University, 1992.

[CSP] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

[Filinski] Andrzej Filinski. *Declarative Continuations and Categorical Duality*. Master's thesis, Computer Science Department, University of Copenhagen (August 1989). DIKU Report 89/11.

[Engines] R. Kent Dybvig and Robert Hieb. Engines from Continuations. *Computer Languages* 14(2):109–123, 1989.

[Jsys] Digital Equipment Corporation. *TOPS-20 Monitor Calls Reference Manual*. January 1980. Document AA-4166D-TM.

[Philbin] James Philbin. *The Design of an Operating System for Modern Programming Languages.* Doctoral Dissertation, Yale University. Yale Computer Science Technical Report YALE/DCS/tr997. May 1993.

[R4RS] J. Rees and W. Clinger (editors). The revised^4 report on the algorithmic language Scheme. *Lisp Pointers* IV(3):1-55, July–September 1991.

[S48] Richard A. Kelsey and Jonathan A. Rees. A tractable Scheme implementation. To appear, *Lisp and Symbolic Computation*, Kluwer Academic Publishers, The Netherlands. (Also URL `ftp://altdorf.ai.mit.edu/pub/jar/lsc.ps`)

[scsh] Olin Shivers. A Scheme shell. To appear in the *Journal of Lisp and Symbolic Computation*. (Also available as technical report TR-635, Laboratory for Computer Science, MIT; and technical report TR-94-10, Department of Computer Science, University of Hong Kong.)

[STNG] Suresh Jagannathan and Jim Philbin. “A foundation for an efficient multi-threaded Scheme System.”

[Wand] Mitchell Wand. “Continuation-based multiprocessing.” In *Conference Record of the 1980 LISP Conference*, pages 19–28, Palo Alto, CA, 1980, J. Allen, editor. The Lisp Company. Republished by ACM.

Errata

I have seized the opportunity to make one emendation to the original cited on the title page: the three occurrences of existential types ($\exists\tau.t$) in sections seven and eight are incorrectly given as universal types ($\forall\tau.t$) in the original paper.