

Unstable-Funcons-beta: Multithreading *

The PPlanCompS Project

Multithreading.cbs | PLAIN | PRETTY

OUTLINE

- Multithreading
 - Initialisation
 - Activation
 - Execution
 - Termination
 - Scheduling
 - Entities
 - Initialisation
 - Activation
 - Execution
 - Termination
 - Scheduling

Multithreading

- [*Datatype* thread-ids
- Datatype* threads
- Funcon* thread-joinable
- Funcon* thread-detached]

Initialisation

- [*Funcon* initialise-multithreading]

Activation

- [*Funcon* multithread
- Funcon* thread-activate
- Funcon* thread-detach]

Execution

- [*Funcon* current-thread
- Funcon* thread-atomic
- Funcon* thread-yield
- Funcon* thread-spin
- Funcon* thread-suspend
- Funcon* thread-resume]

*Suggestions for improvement: plancomps@gmail.com.
Reports of issues: <https://github.com/plancomps/CBS-beta/issues>.

Termination

```
[ Funcon  thread-terminate
  Funcon  is-thread-terminated
  Funcon  thread-value
  Funcon  thread-join
  Funcon  thread-exterminate ]
```

Scheduling

```
[ Funcon  update-thread-stepping
  Funcon  update-thread-schedule
  Funcon  current-thread-schedule
  Funcon  is-thread-preemptible
  Funcon  thread-preemptible
  Funcon  thread-cooperative ]
```

A thread consists of code that can be executed concurrently with code of other threads. The progress of the execution of the threads may be determined cooperatively by threads themselves, or by a preemptive scheduler (or by both).

Threads can be activated by the initial program or by other threads, and share access to the same entities (in contrast to processes). Synchronisation between threads can avoid interference (e.g., data races) arising from concurrent shared access. Synchronisation can also ensure *sequential consistency* of threads, where any concurrent execution corresponds to some (possibly unfair) interleaving of their separate execution steps. Synchronisation can be achieved by various kinds of locks and notifications.

A thread is a value containing a thunk. When a thread has been activated, execution of the body of the thunk can be interleaved with that of other activated threads, possibly subject to scheduling and synchronisation.

Activation of a thread generates a fresh thread-id for referring to it:

Auxiliary Datatype `thread-ids ::= thread-id(_ : atoms)`

A thread is called joinable when other threads can wait for its termination. A joinable thread contains a list of the threads that are currently waiting to join it. A non-joinable thread is called detached:

Auxiliary Datatype `threads ::= thread(_ : thunks(values), _ : (lists thread-ids)?)`

Funcon `thread-joinable(TH : thunks(values)) : ⇒ threads`
`↪ thread(TH, [])`

Funcon `thread-detached(TH : thunks(values)) : ⇒ threads`
`↪ thread(TH)`

In names of funcons and entities, and in informal explanations, a reference to a ‘thread’ generally concerns a thread-id, rather than a value of type `threads`. The only funcons that compute `threads` are `thread-joinable(TH)` and `thread-detached(TH)`; `thread-activate(THR)` is the only funcon that takes an argument of type `threads`.

The thunk contained in a thread can be formed directly from a computation X by `thunk closure` X . Alternatively, `supply(F, V)` forms a thunk by supplying an argument value V to a function F without executing F (in contrast to `apply(F, V)`, which forces the evaluation of the body of F).

The current state of a thread is either active, suspended, terminated, or deleted.

- Activation makes a thread active.

- An active thread can be suspended, and subsequently resumed.
- An active or suspended thread can be terminated.
- A thread can be deleted.

The following mutable entities represent the states of all activated threads.

Entities The `thread-map` contains all active and suspended threads:

```
Entity  ⟨_, thread-map(_ : maps(thread-ids, threads))⟩ →
        ⟨_, thread-map(_ : maps(thread-ids, threads))⟩
```

The `active-thread-set` distinguishes between active and suspended threads:

```
Entity  ⟨_, active-thread-set(_ : sets(thread-ids))⟩ →
        ⟨_, active-thread-set(_ : sets(thread-ids))⟩
```

```
Funcon  is-some-thread-active : ⇒ booleans
```

```
Rule    ⟨is-some-thread-active, active-thread-set(ATS)⟩ →
        not is-equal(ATS, { })
```

```
Funcon  is-some-thread-suspended : ⇒ booleans
```

```
Rule    ⟨is-some-thread-suspended, active-thread-set(ATS), thread-map(TM)⟩ →
        not is-equal(ATS, dom TM)
```

The `thread-stepping` entity identifies the thread whose step is currently being executed. When defined, it identifies an active thread:

```
Entity  ⟨_, thread-stepping(_ : thread-ids?)⟩ →
        ⟨_, thread-stepping(_ : thread-ids?)⟩
```

The `terminated-thread-map` contains the values computed by terminated joinable threads:

```
Entity  ⟨_, terminated-thread-map(_ : maps(thread-ids, values))⟩ →
        ⟨_, terminated-thread-map(_ : maps(thread-ids, values))⟩
```

Deleted threads are not contained in any of the above entities. Terminated detached threads are always deleted.

Scheduling information for each thread may affect the interleaving of thread steps:

```
Entity  ⟨_, thread-schedule(_ : sets(ground-values))⟩ →
        ⟨_, thread-schedule(_ : sets(ground-values))⟩
```

Scheduling can change dynamically between preemptive and cooperative.

Initialisation The entities used to model multithreading need initialising:

```
Funcon  initialise-multithreading : ⇒ null-type
        ~⇒ sequential(
            initialise-thread-map,
            initialise-active-thread-set,
            initialise-thread-stepping,
            initialise-terminated-thread-map,
            initialise-thread-schedule)
```

The initial values are generally quite obvious:

Auxiliary Funcon `initialise-thread-map` : \Rightarrow null-type

Rule `initialise-thread-map` \rightarrow
 $\langle \text{null-value}, \text{thread-map}(\text{map}(\)) \rangle$

Auxiliary Funcon `initialise-active-thread-set` : \Rightarrow null-type

Rule `initialise-active-thread-set` \rightarrow
 $\langle \text{null-value}, \text{active-thread-set}\{ \} \rangle$

Auxiliary Funcon `initialise-thread-stepping` : \Rightarrow null-type

Rule `initialise-thread-stepping` \rightarrow
 $\langle \text{null-value}, \text{thread-stepping}(\) \rangle$

Auxiliary Funcon `initialise-terminated-thread-map` : \Rightarrow null-type

Rule `initialise-terminated-thread-map` \rightarrow
 $\langle \text{null-value}, \text{terminated-thread-map}(\text{map}(\)) \rangle$

Auxiliary Funcon `initialise-thread-schedule` : \Rightarrow null-type

Rule `initialise-thread-schedule` \rightarrow
 $\langle \text{null-value}, \text{thread-schedule}\{ \} \rangle$

Activation Multithreading can start by activating a single thread, which can then activate further threads:

Funcon `multithread`($_ : \Rightarrow \text{values}$) : $\Rightarrow \text{values}$

`multithread` X forms a joinable thread from `thunk closure` X and activates it. On normal termination of all threads, it gives the value computed by X . On abrupt termination (caused by a thread step, or by the failure of the check that all threads have terminated) the reason for it:

Rule `multithread` $X \rightsquigarrow$
`sequential`(
 `initialise-multithreading`,
 `give`(
 `thread-activate thread-joinable thunk closure` X ,
 `handle-abrupt`(
 `sequential`(
 `while-true`(
 `is-some-thread-active`,
 `sequential`(`update-thread-stepping`, `thread-step`)),
 `check not is-some-thread-suspended`,
 `thread-value given`),
 `given`)))

The **update-thread-stepping** funcon determines the thread-id for the next step, which may depend on whether the previous step has yielded, on its preemptibility, and on the current scheduling.

A thread can activate the execution of thread-activates:

Funcon **thread-activate**($_ : \text{threads}$) : $\Rightarrow \text{thread-ids}$

When a thread is activated, a fresh thread-id is generated, and the thread is included in the thread-map and the active-thread-set:

$$\begin{array}{c} \text{thread-id}(\text{fresh-atom}) \longrightarrow TI \\ \text{map-unite}(\{TI \mapsto THR\}, TM) \rightsquigarrow TM' \\ \text{set-unite}(\{TI\}, ATS) \rightsquigarrow ATS' \\ \hline \text{Rule} \quad \langle \text{thread-activate}(THR : \text{threads}), \text{thread-map}(TM), \text{active-thread-set}(ATS) \rangle \longrightarrow \\ \langle TI, \text{thread-map}(TM'), \text{active-thread-set}(ATS') \rangle \end{array}$$

A joinable thread can be detached after its activation, discarding its list of joining threads:

Funcon **thread-detach**($_ : \text{thread-ids}$) : $\Rightarrow \text{null-type}$

When the thread has not yet terminated, it remains in the thread-map:

$$\begin{array}{c} \text{map-lookup}(TM, TI) \rightsquigarrow \text{thread}(TH, _) \\ \text{map-override}(\{TI \mapsto \text{thread}(TH)\}, TM) \rightsquigarrow TM' \\ \hline \text{Rule} \quad \langle \text{thread-detach}(TI : \text{thread-ids}), \text{thread-map}(TM) \rangle \longrightarrow \\ \langle \text{null-value}, \text{thread-map}(TM') \rangle \end{array}$$

When the thread has already terminated, detaching it deletes it from the terminated-thread-map:

$$\begin{array}{c} \text{is-in-set}(TI, \text{dom } TMV) == (\text{true}) \\ \text{map-delete}(TMV, \{TI\}) \rightsquigarrow TMV' \\ \hline \text{Rule} \quad \langle \text{thread-detach}(TI : \text{thread-ids}), \text{terminated-thread-map}(TMV) \rangle \longrightarrow \\ \langle \text{null-value}, \text{terminated-thread-map}(TMV') \rangle \end{array}$$

(A funcon for making a detached thread joinable could be defined similarly.)

Execution The thread-stepping is undefined only before multithreading starts, and when all non-terminated threads are suspended.

Funcon **current-thread** : $\Rightarrow \text{thread-ids}$

Rule $\langle \text{current-thread}, \text{thread-stepping}(TI) \rangle \longrightarrow TI$

current-thread is only intended for use in threads:

Rule $\langle \text{current-thread}, \text{thread-stepping}(_) \rangle \longrightarrow \text{fail}$

Stepping The funcon **thread-step** executes a single step of the thread identified by **thread-stepping**:

Auxiliary Funcon **thread-step** : $\Rightarrow \text{null-type}$

Whenever THR executes a step and THR' represents the remaining steps, the thread-map is updated to map TI to THR' .

If the body of the thread can make a step, so can **thread-step**:

$$\begin{array}{l}
 \text{lookup}(TM, TI) \rightsquigarrow \text{thread}(\text{thunk abstraction}(X), L?) \\
 \langle X, \text{thread-stepping}(TI), \text{thread-map}(TM) \rangle \longrightarrow \\
 \langle X', \text{thread-stepping}(TI?), \text{thread-map}(TM') \rangle \\
 \text{if-true-else}(\text{is-in-set}(TI, \text{dom } TM'), \\
 \text{map-override}(\{TI \mapsto \text{thread}(\text{thunk abstraction}(X'), L?)\}, TM'), \\
 TM') \rightsquigarrow \\
 TM'' \\
 \text{Rule} \quad \frac{}{\langle \text{thread-step}, \text{thread-stepping}(TI), \text{thread-map}(TM) \rangle \longrightarrow \langle \text{null-value}, \text{thread-stepping}(TI?), \text{thread-map}(TM'') \rangle}
 \end{array}$$

The only other case for a next step is when a thread has terminated normally, and is to be removed from the executing thread map.

If it is detached, its computed value is discarded:

$$\begin{array}{l}
 \text{lookup}(TM, TI) \rightsquigarrow \text{thread}(\text{thunk abstraction}(X)) \\
 X \rightsquigarrow (V : \text{values}) \\
 \text{map-delete}(TM, \{TI\}) \rightsquigarrow TM' \\
 \text{set-difference}(ATS, \{TI\}) \rightsquigarrow ATS' \\
 \text{Rule} \quad \frac{}{\langle \text{thread-step}, \text{active-thread-set}(ATS), \text{thread-stepping}(TI), \text{thread-map}(TM) \rangle \longrightarrow \langle \text{null-value}, \text{active-thread-set}(ATS'), \text{thread-stepping}(), \text{thread-map}(TM') \rangle}
 \end{array}$$

If the thread is joinable, all its joining threads are resumed, and its computed value is made available in the terminated-thread-map:

$$\begin{array}{l}
 \text{lookup}(TM, TI) \rightsquigarrow \text{thread}(\text{thunk abstraction}(X), [TI*]) \\
 X \rightsquigarrow (V : \text{values}) \\
 \text{map-delete}(TM, \{TI\}) \rightsquigarrow TM' \\
 \text{set-unite}(\text{set-difference}(ATS, \{TI\}), \{TI*\}) \rightsquigarrow ATS' \\
 \text{map-unite}(TVM, \{TI \mapsto V\}) \rightsquigarrow TVM' \\
 \text{Rule} \quad \frac{}{\langle \text{thread-step}, \text{active-thread-set}(ATS), \text{thread-stepping}(TI), \text{thread-map}(TM), \text{terminated-thread-map}(TVM) \rangle \longrightarrow \langle \text{null-value}, \text{active-thread-set}(ATS'), \text{thread-stepping}(), \text{thread-map}(TM'), \text{terminated-thread-map}(TVM') \rangle}
 \end{array}$$

Sync atomicity **thread-atomic**(X) computes X in a single transition. Assuming that the potentially interfering effects of X may only be updates on the store, thread suspension and/or resumption, and abrupt termination, and that X always terminates, the outcome of **thread-atomic**(X) is the same as that of computing X without preemption.

(The funcon **atomic**(X) generalises **thread-atomic**(X) to allow X with arbitrary effects. When the CBS notation “ \longrightarrow_1 ; \longrightarrow_2 ” for composing transitions has been implemented by the interpreter generation tools, uses of **thread-atomic**(X) are to be replaced by **atomic**(X).)

Auxiliary Funcon **thread-atomic**($_ : \Rightarrow \text{values}$) : $\Rightarrow \text{values}$

$$\begin{array}{l}
\text{Rule} \quad \frac{\langle X, \text{store}(\sigma), \text{active-thread-set}(ATS), \text{thread-stepping}(TI) \rangle \xrightarrow{\text{abrupted}(\)} \langle X', \text{store}(\sigma'), \text{active-thread-set}(ATS'), \text{thread-stepping}(TI') \rangle}{\langle \text{thread-atomic}(X'), \text{store}(\sigma'), \text{active-thread-set}(ATS'), \text{thread-stepping}(TI') \rangle \xrightarrow{\text{abrupted}(\)} \langle V, \text{store}(\sigma''), \text{active-thread-set}(ATS''), \text{thread-stepping}(TI''?) \rangle} \\
\text{Rule} \quad \frac{\langle \text{thread-atomic}(X), \text{store}(\sigma), \text{active-thread-set}(ATS), \text{thread-stepping}(TI) \rangle \xrightarrow{\text{abrupted}(\)} \langle V, \text{store}(\sigma''), \text{active-thread-set}(ATS''), \text{thread-stepping}(TI''?) \rangle}{\langle \text{thread-atomic}(X), \text{store}(\sigma), \text{active-thread-set}(ATS), \text{thread-stepping}(TI) \rangle \xrightarrow{\text{abrupted}(A)} \langle V, \text{store}(\sigma''), \text{active-thread-set}(ATS''), \text{thread-stepping}(TI''?) \rangle} \\
\text{Rule} \quad \frac{X \xrightarrow{\text{abrupted}(A)} X'}{\text{thread-atomic}(X) \xrightarrow{\text{abrupted}(A)} \text{thread-atomic}(X')} \\
\text{Rule} \quad \frac{X \xrightarrow{\text{abrupted}(\)} (V : \text{values})}{\text{thread-atomic}(X) \xrightarrow{\text{abrupted}(\)} V} \\
\text{Rule} \quad \frac{X \xrightarrow{\text{abrupted}(A)} (V : \text{values})}{\text{thread-atomic}(X) \xrightarrow{\text{abrupted}(A)} \text{thread-atomic}(V)} \\
\text{Rule} \quad \text{thread-atomic}(V : \text{values}) \longrightarrow V
\end{array}$$

Note that if the execution of `thread-atomic(X)` involves `thread-yield()` or `thread-suspend()`, this makes thread-stepping undefined, so it has to be the final step of `X`.

Yielding A thread can yield execution:

Funcon `thread-yield(_ : thread-ids?) : \Rightarrow null-type`

When the argument thread-id is omitted, the next thread to be executed is left undefined, to be determined by the scheduler:

$$\begin{array}{l}
\text{Rule} \quad \text{thread-yield}(\) \longrightarrow \\
\quad \langle \text{null-value}, \text{thread-stepping}(\) \rangle
\end{array}$$

When the argument thread-id is `TI`, it has to be an active thread:

$$\begin{array}{l}
\text{Rule} \quad \frac{\text{is-in-set}(TI, ATS) == (\text{true})}{\langle \text{thread-yield}(TI : \text{thread-ids}), \text{active-thread-set}(ATS) \rangle \longrightarrow \langle \text{null-value}, \text{thread-stepping}(TI) \rangle} \\
\text{Rule} \quad \frac{\text{is-in-set}(TI, ATS) == (\text{false})}{\langle \text{thread-yield}(TI : \text{thread-ids}), \text{active-thread-set}(ATS) \rangle \longrightarrow \text{fail}}
\end{array}$$

`thread-spin(X)` repeatedly executes `X` while it fails, allowing interleaving with other threads.

Funcon `thread-spin(X : \Rightarrow values) : \Rightarrow values`
 \rightsquigarrow `else(`
 $\quad X,$
 $\quad \text{sequential}(\text{thread-yield}(\), \text{thread-spin}(X)))$

Suspension and resumption A thread may suspend one or more threads that are currently being executed:

$$\begin{array}{l}
\text{Funcon } \text{thread-suspend}(_ : \text{thread-ids}^+) : \Rightarrow \text{null-type} \\
\text{is-in-set}(TI, \{TI^+\}) == (\text{false}) \\
\text{is-subset}(\{TI^+\}, ATS) == (\text{true}) \\
\text{set-difference}(ATS, \{TI^+\}) \rightsquigarrow ATS' \\
\text{Rule } \frac{}{\langle \text{thread-suspend}(TI^+ : \text{thread-ids}^+), \text{thread-stepping}(TI), \text{active-thread-set}(ATS) \rangle \longrightarrow \langle \text{null-value}, \text{thread-stepping}(TI), \text{active-thread-set}(ATS') \rangle}
\end{array}$$

If TI^+ includes the current thread, suspension is accompanied by yielding:

$$\begin{array}{l}
\text{is-in-set}(TI, \{TI^+\}) == (\text{true}) \\
\text{is-subset}(\{TI^+\}, ATS) == (\text{true}) \\
\text{set-difference}(ATS, \{TI^+\}) \rightsquigarrow ATS' \\
\text{Rule } \frac{}{\langle \text{thread-suspend}(TI^+ : \text{thread-ids}^+), \text{thread-stepping}(TI), \text{active-thread-set}(ATS) \rangle \longrightarrow \langle \text{null-value}, \text{thread-stepping}(_), \text{active-thread-set}(ATS') \rangle}
\end{array}$$

(Deadlock occurs if the last non-suspended thread suspends itself.)

A thread may resume any number of suspended threads:

$$\begin{array}{l}
\text{Funcon } \text{thread-resume}(_ : \text{thread-ids}^*) : \Rightarrow \text{null-type} \\
\text{is-in-set}(TI, \{TI^*\}) == (\text{false}) \\
\text{set-intersect}(ATS, \{TI^*\}) == \{ \} \\
\text{set-unite}(ATS, \{TI^*\}) \rightsquigarrow ATS' \\
\text{Rule } \frac{}{\langle \text{thread-resume}(TI^* : \text{thread-ids}^*), \text{thread-stepping}(TI), \text{active-thread-set}(ATS) \rangle \longrightarrow \langle \text{null-value}, \text{thread-stepping}(TI), \text{active-thread-set}(ATS') \rangle}
\end{array}$$

Termination If the thread-map becomes empty, and there are no suspended threads, the entire multithread computation terminates normally. If it becomes empty while suspended threads remain to be executed, this is regarded as deadlock, and the computation fails.

If the execution of an individual thread terminates normally, the body of its thunk gives its computed value. The thread is removed from the thread-map, and the computed value is added to the terminated-thread-map.

Abrupt termination of a thread body causes immediate abrupt termination of the entire multithread computation. This can be avoided by wrapping the bodies of all threads in appropriate handlers for abrupt termination.

A thread can terminate itself or another thread, optionally specifying its computed value:

$$\text{Funcon } \text{thread-terminate}(_ : \text{thread-ids}, _ : \text{values}^?) : \Rightarrow \text{null-type}$$

When the thread is detached, no value is specified:

$$\begin{array}{l}
\text{lookup}(TM, TI') \rightsquigarrow \text{thread}(\text{thunk abstraction } X) \\
\text{map-delete}(TM, \{TI'\}) \rightsquigarrow TM' \\
\text{set-difference}(ATS, \{TI'\}) \rightsquigarrow ATS' \\
\text{when-true}(\text{not is-equal}(TI, TI'), TI) \rightsquigarrow TI? \\
\text{Rule } \frac{}{\langle \text{thread-terminate}(TI' : \text{thread-ids}), \text{thread-stepping}(TI), \text{thread-map}(TM), \text{active-thread-set}(ATS) \rangle \longrightarrow \langle \text{null-value}, \text{thread-stepping}(TI?), \text{thread-map}(TM'), \text{active-thread-set}(ATS') \rangle}
\end{array}$$

When the thread is joinable, its value has to be specified:

$$\begin{array}{c}
\text{lookup}(TM, TI') \rightsquigarrow \text{thread}(\text{thunk abstraction } X, [TI^*]) \\
\text{map-delete}(TM, \{TI'\}) \rightsquigarrow TM' \\
\text{set-unite}(\text{set-difference}(ATS, \{TI'\}), \{TI^*\}) \rightsquigarrow ATS' \\
\text{map-unite}(TVM, \{TI' \mapsto V\}) \rightsquigarrow TVM' \\
\text{when-true}(\text{not is-equal}(TI, TI'), TI) \rightsquigarrow TI? \\
\hline
\text{Rule} \quad \langle \text{thread-terminate}(TI' : \text{thread-ids}, V : \text{values}), \text{thread-stepping}(TI), \\
\text{thread-map}(TM), \text{terminated-thread-map}(TVM), \text{active-thread-set}(ATS) \rangle \longrightarrow \\
\langle \text{null-value}, \text{thread-stepping}(TI?), \\
\text{thread-map}(TM'), \text{terminated-thread-map}(TVM'), \text{active-thread-set}(ATS') \rangle
\end{array}$$

A thread can test whether a joinable thread has terminated:

$$\begin{array}{c}
\text{Funcon} \quad \text{is-thread-terminated}(_ : \text{thread-ids}) : \Rightarrow \text{booleans} \\
\text{Rule} \quad \langle \text{is-thread-terminated}(TI : \text{thread-ids}), \text{terminated-thread-map}(TVM) \rangle \longrightarrow \\
\text{is-value}(\text{map-lookup}(TVM, TI))
\end{array}$$

If so, it can get the computed value:

$$\begin{array}{c}
\text{Funcon} \quad \text{thread-value}(_ : \text{thread-ids}) : \Rightarrow \text{values} \\
\text{Rule} \quad \langle \text{thread-value}(TI : \text{thread-ids}), \text{terminated-thread-map}(TVM) \rangle \longrightarrow \\
\text{checked map-lookup}(TVM, TI)
\end{array}$$

Joining a thread may cause suspension:

$$\begin{array}{c}
\text{Funcon} \quad \text{thread-join}(_ : \text{thread-ids}) : \Rightarrow \text{null-type} \\
\text{lookup}(TM, TI') \rightsquigarrow \text{thread}(TH, [TI^*]) \\
\text{map-override}(\{TI' \mapsto \text{thread}(TH, [TI^*, TI])\}, TM) \rightsquigarrow TM' \\
\text{set-difference}(ATS, \{TI\}) \rightsquigarrow ATS' \\
\hline
\text{Rule} \quad \langle \text{thread-join}(TI' : \text{thread-ids}), \text{thread-map}(TM), \text{thread-stepping}(TI), \\
\text{active-thread-set}(ATS) \rangle \longrightarrow \\
\langle \text{null-value}, \text{thread-map}(TM'), \\
\text{thread-stepping}(_), \text{active-thread-set}(ATS') \rangle
\end{array}$$

If a joinable thread has already terminated, the `terminated-thread-map` holds its value:

$$\begin{array}{c}
\text{Rule} \quad \frac{\text{is-value}(\text{lookup}(TVM, TI')) == \text{true}}{\langle \text{thread-join}(TI' : \text{thread-ids}), \text{terminated-thread-map}(TVM) \rangle \longrightarrow \text{null-value}}
\end{array}$$

Trying to join a detached thread fails:

$$\begin{array}{c}
\text{Rule} \quad \frac{\text{lookup}(TM, TI') \rightsquigarrow \text{thread}(TH)}{\langle \text{thread-join}(TI' : \text{thread-ids}), \text{thread-map}(TM) \rangle \longrightarrow \text{fail}} \\
\\
\text{Rule} \quad \frac{\text{lookup}(TM, TI') == () \quad \text{lookup}(TVM, TI') == ()}{\langle \text{thread-join}(TI' : \text{thread-ids}), \text{thread-map}(TM), \text{terminated-thread-map}(TVM) \rangle \longrightarrow \text{fail}}
\end{array}$$

Extermination of a thread both terminates it and prevents its subsequent inspection.

$$\begin{array}{l}
\text{Funcon } \text{thread-exterminate}(_ : \text{thread-ids}) : \Rightarrow \text{null-type} \\
\quad TI' \neq TI \\
\quad \text{map-delete}(TM, \{TI'\}) \rightsquigarrow TM' \\
\quad \text{map-delete}(TVM, \{TI'\}) \rightsquigarrow TVM' \\
\quad \text{set-difference}(ATS, \{TI'\}) \rightsquigarrow ATS' \\
\text{Rule } \frac{}{\langle \text{thread-exterminate}(TI' : \text{thread-ids}), \text{thread-map}(TM), \text{thread-stepping}(TI), \\
\quad \text{terminated-thread-map}(TVM), \text{active-thread-set}(ATS) \rangle \longrightarrow \\
\quad \langle \text{null-value}, \text{thread-map}(TM'), \text{thread-stepping}(TI), \\
\quad \text{terminated-thread-map}(TVM'), \text{active-thread-set}(ATS') \rangle}
\end{array}$$

Scheduling A scheduler determines the interleaving of thread execution, based on mutable information regarding features such as preemptibility, priority, and time-sharing.

The next thread scheduled for execution is an active thread, or undefined when there are no active threads. It may be the same thread that last made a step.

(The following definitions allow `update-thread-stepping` to update `thread-stepping` to any valid thread. They are to be replaced by declarations of built-in funcons, allowing exploration of different interleavings using oracles.)

$$\text{Funcon } \text{update-thread-stepping} : \Rightarrow \text{null-type}$$

When `thread-stepping` is TI and that thread is not preemptible, `update-thread-stepping` has no effect:

$$\text{Rule } \frac{\text{is-thread-preemptible}(TI) \longrightarrow \text{false}}{\langle \text{update-thread-stepping}, \text{thread-stepping}(TI) \rangle \longrightarrow \text{null-value}}$$

When `thread-stepping` is TI and that thread is preemptible, or when `thread-stepping` is undefined, `update-thread-stepping` may set it to any active TI' :

$$\begin{array}{l}
\text{Rule } \frac{\text{is-thread-preemptible}(TI) \longrightarrow (\text{true}) \quad \text{some-element}(ATS) \rightsquigarrow TI'}{\langle \text{update-thread-stepping}, \text{thread-stepping}(TI), \text{active-thread-set}(ATS) \rangle \longrightarrow \langle \text{null-value}, \text{thread-stepping}(TI') \rangle} \\
\text{Rule } \frac{\text{some-element}(ATS) \rightsquigarrow TI'}{\langle \text{update-thread-stepping}, \text{thread-stepping}(_), \text{active-thread-set}(ATS) \rangle \longrightarrow \langle \text{null-value}, \text{thread-stepping}(TI') \rangle}
\end{array}$$

When there are no active threads, `update-thread-stepping` ensures that `thread-stepping` is undefined:

$$\text{Rule } \langle \text{update-thread-stepping}, \text{thread-stepping}(TI?), \text{active-thread-set}\{ \} \rangle \longrightarrow \langle \text{null-value}, \text{thread-stepping}(_) \rangle$$

Scheduling information for each thread can be inspected and updated:

$$\begin{array}{l}
\text{Funcon } \text{update-thread-schedule}(_ : \text{sets}(\text{ground-values})) : \Rightarrow \text{null-type} \\
\text{Rule } \text{update-thread-schedule}(VS : \text{sets}(\text{ground-values})) \longrightarrow \langle \text{null-value}, \text{thread-schedule}(VS) \rangle
\end{array}$$

Funcon `current-thread-schedule` : \Rightarrow `sets(ground-values)`
Rule $\langle \text{current-thread-schedule}, \text{thread-schedule}(VS) \rangle \longrightarrow VS$

Datatype `thread-preemptibilities` ::= `thread-preemptible` | `thread-cooperative`

Funcon `is-thread-preemptible`(`_` : `thread-ids`) : \Rightarrow `booleans`
 \rightsquigarrow `not is-in-set(thread-cooperative, current-thread-schedule)`

For now, all threads are preemptible unless the scheduling includes cooperative.

The representation of scheduling information is left open here, together with the details of how it affects the result of `update-thread-stepping`.