Unstable-Funcons-beta: Notifications *

The PLanCompS Project

Notifications.cbs | PLAIN | PRETTY

OUTLINE

Notifications

```
[ Barriers
  Funcon  barrier-create
  Funcon  barrier-sync
  Funcon  barrier-sync-else-wait
  Conditions
  Funcon  condition-create
  Funcon  condition-wait
  Funcon  condition-wait-with-lock
  Funcon  condition-notify-all
  Funcon  condition-notify-first
  Rendezvous
  Funcon  rendezvous-create
  Funcon  rendezvous-sync
  Funcon  rendezvous-sync-else-wait ]
```

Threads may synchronise by waiting for notifications. In contrast to locks, notifications are ephemeral, and do not get held and released.

Barriers A barrier notifies all requesting threads when a specified number of requests for it have been made. Subsequent requests give immediate notification.

```
Funcon  barrier-create(N : pos-ints) : ⇒ syncs
          ⤳ give(
                sync-create(
                  sync-feature-create sync-waiting-list,
                  sync-feature-create sync-count),
                sequential(
                  assign(sync-feature(given, sync-count), N),
                  given))
```

When the barrier is already open, requests to pass it are granted immediately. When the barrier is closed, and only one more thread needs to arrive, granting a request for it opens the barrier and resumes all the threads wiating for it; otherwise the request fails.

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

```
Funcon  barrier-sync(SY : syncs) : ⇒ null-type
          ⇝ thread-atomic give(
                sync-feature(SY, sync-count),
                else(
                  check-true is-equal(assigned given, 0),
                  sequential(
                    check-true is-equal(assigned given, 1),
                    assign(given, 0),
                    thread-resume list-elements
                      assigned sync-feature(SY, sync-waiting-list),
                    assign(sync-feature(SY, sync-waiting-list), [ ]))))
```

When the request fails, the current thread is added to the waiting list, and suspended until the request can be granted:

```
Funcon  barrier-sync-else-wait(SY : syncs) : ⇒ null-type
          ⇝ thread-atomic else(
                barrier-sync(SY),
                sequential(
                  sync-waiting-list-add(SY, current-thread),
                  assign(
                    sync-feature(SY, sync-count),
                    checked nat-pred assigned sync-feature(SY, sync-count)),
                  thread-suspend current-thread))
```

Conditions A condition is used to represent whether some property holds or not. Threads may request to be notified when another thread makes the property hold.

A condition may notify either one or all of its requesting threads. When it has to notify one thread but more than one request for notification has been made, the choice of thread may be determined by the scheduler. When it has to notify more than one thread, the property associated with the condition may have been invalidated by the time the executions of some of them are resumed, and threads may need to iterate requests for notifications.

In practice, a condition is generally associated with an exclusive lock. When a thread awaiting the condition is notified, it requests the exclusive lock and tests whether the required property holds; if it does not, the thread releases the exclusive lock, and atomically reverts to requesting the notification.

```
Funcon  condition-create : ⇒ syncs
          ⇝ sync-create(
                sync-feature-create sync-waiting-list)
```

A condition request always adds the current thread to the waiting list, and suspends it until the request can be granted. (In practice, it takes also an associated exclusive lock as a further argument, assumed to be held by the current thread, and releases it at the same time as suspending the thread.)

```
Funcon  condition-wait(SY : syncs) : ⇒ null-type
          ⇝ thread-atomic sequential(
                sync-waiting-list-add(SY, current-thread),
                thread-suspend current-thread)
```

In practice, a condition request usually takes also an associated exclusive lock as a further argument, assumed to be held by the current thread, releases it together with suspending the thread, and waits for the lock when resumed:

```
Funcon  condition-wait-with-lock(SY : syncs, L : syncs) : ⇒ null-type
          ⇝ sequential(
                thread-atomic sequential(
                  exclusive-lock-release(L),
                  sync-waiting-list-add(SY, current-thread),
                  thread-suspend current-thread),
                exclusive-lock-sync-else-wait(L))
```

Threads that are waiting for the condition are notified simply by resuming them. To notify them all:

```
Funcon  condition-notify-all(SY : syncs) : ⇒ null-type
          ⇝ thread-atomic
                sequential(
                  thread-resume list-elements
                    assigned sync-feature(SY, sync-waiting-list),
                  assign(sync-feature(SY, sync-waiting-list), [ ]))
```

To notify just one of the waiting threads:

```
Funcon  condition-notify-first(SY : syncs) : ⇒ null-type
          ⇝ thread-atomic
                give(
                  sync-waiting-list-head-remove(SY),
                  thread-resume given)
```

Rendezvous A rendezvous notifies all requesting threads as soon as a specified number N of them have made *matching* requests for it. The rendezvous can store any number of non-matching requests. If a request that completes a rendezvous matches different sets of N-1 pending requests, the 'lexicographically' earliest set of requests is selected; for a binary rendezvous, this is the first matching request in the stored list.

```
Funcon  rendezvous-create(N : pos-ints) : ⇒ syncs
          ⇝ give(
                sync-create(
                  sync-feature-create sync-waiting-list,
                  sync-feature-create sync-count),
                sequential(
                  assign(sync-feature(given, sync-count), N),
                  given))
```

Each rendezvous request includes a pattern, and the corresponding notifications give environments obtained by matching the patterns against the same unified value. When the pattern in each request is simply a value, a rendezvous notifies all the requesting threads as soon as the specified number of requests with the same value have been made. When the pattern in one request is a value, a pattern in another request may bind an identifier to that value, giving one-way data flow.

A rendezvous request may also include a set of additional threads which are all required to participate in the rendezvous. When a pair of matching binary rendezvous requests each specify the other thread as the only required participant, the rendezvous is restricted to that pair of threads. When one of the sets is empty, the rendezvous may involve any other thread.

> In this simplified version, rendezvous are always binary, patterns in requests are ground values, and sets of required threads are omitted.

When a rendezvous is available, granting a request for it removes the first matching element from the waiting list, and resumes its thread; otherwise the request fails.

```
Funcon   rendezvous-sync(SY : syncs, V : ground-values) : ⇒ null-type
           ⇝ give(
                 assigned sync-feature(SY, sync-waiting-list),
                 sequential(
                    check-true is-rendezvous-match(given, V),
                    assign(
                       sync-feature(SY, sync-waiting-list),
                       rendezvous-first-match-drop(given, V)),
                    thread-resume
                       rendezvous-first-match-thread(given, V)))
```

When the request fails, a tuple of the value and the current thread is added to the waiting list, and the thread suspended until the request can be granted:

```
Funcon   rendezvous-sync-else-wait(SY : syncs, V : ground-values) : ⇒ null-type
           ⇝ thread-atomic else(
                 rendezvous-sync(SY, V),
                 sequential(
                    sync-waiting-list-add(SY, tuple(V, current-thread)),
                    thread-suspend current-thread))
```

The remaining rendezvous funcons are all auxiliary:

```
Auxiliary Type   rendezvous-waits ⇝ tuples(ground-values, thread-ids)
```

The funcon is-rendezvous-match(L, V) returns whether the list L contains tuple(V, TI) for some TI:

```
Auxiliary Funcon   is-rendezvous-match(_ : lists(rendezvous-waits), _ : ground-values) : ⇒ booleans

Rule   is-rendezvous-match([tuple(V′, TI), P*], V : values) ⇝
          if-true-else(is-equal(V′, V), true, is-rendezvous-match([P*], V))
Rule   is-rendezvous-match([ ], V : values) ⇝ false
```

The funcon rendezvous-first-match-thread(L, V) returns the thread-id of the first element of L with value V:

```
Auxiliary Funcon   rendezvous-first-match-thread(_ : lists(rendezvous-waits), _ : values)
                        : ⇒ thread-ids

Rule   rendezvous-first-match-thread([tuple(V′, TI), P*], V : values) ⇝
          if-true-else(is-equal(V′, V), TI, rendezvous-first-match-thread([P*], V))
Rule   rendezvous-first-match-thread([ ], V : values) ⇝ fail
```

The funcon rendezvous-first-match-drop(L, V) returns the list L omitting the first element with value V:

Auxiliary Funcon rendezvous-first-match-drop(_ : lists(rendezvous-waits), _ : values)
  : ⇒ lists(rendezvous-waits)

Rule rendezvous-first-match-drop([tuple(V', TI), P^*], V : values) ⇝
  if-true-else(
    is-equal(V', V),
    [P^*],
    cons(tuple(V', TI), rendezvous-first-match-drop([P^*], V)))

Rule rendezvous-first-match-drop([ ], V : values) ⇝ fail

A series of rendezvous between the same two threads is called an extended rendezvous. After the completion of each rendezvous in the series, one of the threads may immediately request the next, allowing the other thread to execute some code before synchronising. A simple rendezvous is restricted to synchronisation, and does not involve ordinary computation steps.