```
        return NULL;
    } else {
        set_ref_count(k);
        recycle_as_continuation();
        task& t_k   = *new(allocate_child()) T(...); spawn(t_k);
        task& t_k-1 = *new(allocate_child()) T(...); spawn(t_k-1);
        ...
        // Return pointer to first child instead of spawning it,
        // to remove unnecessary overhead.
        task& t_1 = *new(allocate_child()) T(...);
        return &t_1;
    }
}
```

The key points of the pattern are:

- The call to `set_ref_count` uses *k* as its argument. There is no extra +1 as there is in blocking style discussed in Section 10.8.1.
- Each child task is allocated by `allocate_child`.
- The continuation is recycled from the parent, and hence gets the parent's state without doing copy operations.

Deleted: 10.8.1

10.8.2.2 Recycling Parent as a Child

This style is useful when the child inherits much of its state from a parent and the continuation does not need the state of the parent. The child must have the same type as the parent. In the example, C is the type of the continuation, and must derive from class `task`. If C does nothing except wait for all children to complete, then C can be the class `empty_task` (10.4).

Deleted: 10.4

```
task* T::execute() {
    if( not recursing any further ) {
        ...
        return NULL;
    } else {
        // Construct continuation
        C& c = allocate_continuation();
        c.set_ref_count(k);
        // Recycle self as first child
        task& t_k   = *new(c.allocate_child()) T(...); spawn(t_k);
        task& t_k-1 = *new(c.allocate_child()) T(...); spawn(t_k-1);
        ...
        task& t_2 = *new(c.allocate_child()) T(...);  spawn(t_2);
        // task t_1 is our recycled self.
        recycle_as_child_of(c);
        update fields of *this to subproblem to be solved by t_1
        return this;
```

Deleted: set_ref_count(k);¶

```
    }
}
```

The key points of the pattern are:

- The call to set_ref_count uses *k* as its argument. There is no extra 1 as there is in blocking style discussed in Section 10.8.1
- Each child task except for t_1 is allocated by c.allocate_child. It is critical to use c.allocate_child, and not (*this).allocate_child; otherwise the task graph will be wrong.
- Task t_1 is recycled from the parent, and hence gets the parent's state without performing copy operations. Do not forget to update the state to represent a child subproblem; otherwise infinite recursion will occur.

Deleted: 10.8.1

10.8.3 Letting Main Thread Work While Child Tasks Run

Sometimes it is desirable to have the main thread continue execution while child tasks are running. The following pattern does this by using a dummy empty_task (10.4).

Deleted: 10.4

```
task* dummy = new( task::allocate_root() ) empty_task;
dummy->set_ref_count(k+1);
task& t_k = *new( dummy->allocate_child() ) T;  dummy->spawn(t_k);
task& t_k-1= *new( dummy->allocate_child() ) T;  dummy->spawn(t_k-1);
...
task& t_1 = *new( dummy->allocate_child() ) T;  dummy->spawn(t_1);
...do any other work...
dummy->wait_for_all();
dummy->destroy(*dummy);
```

The key points of the pattern are:

- The dummy task is a placeholder and never runs.
- The call to set_ref_count uses *k*+1 as its argument.
- The dummy task must be explicitly destroyed.