Notes on the verification of the concurrent spanning tree algorithm

```
1  let rec span x =
2    if (x == null) then
3      false
4    else
5      if (CAS(x.mark, false, true)) then
6        let (l, r) =
7          (span (x.left) || span (x.right)) in
8        if (¬l) then x.left := null;
9        if (¬r) then x.right := null;
10       true
11     else
12       false
```

Figure 1. The pseudo-code of the spanning tree algorithm in ML style

Abstract

We give a short description of the verification of the concurrent algorithm for computing a spanning tree of a graph described below.

1. The program and intuition

The algorithm verified is depicted in Figure 1. In this text we assume graphs are doubly branching. Therefore, we refer to the children of a node as the left or right child (which may not exist for some nodes). We write $x.left = null$ and $x.right = null$ whenever a node x has no left or right child respectively. Whenever not clear to which graph we are referring we use $g.x.left$ and $g.x.right$ to refer to the children of a node x in a particular graph g.

We write $path(p, g, x, y)$ to say that p is a path in g from x to y. Here a path p is a sequence $p = [d_1, \ldots, d_n]$ where $d_1, \ldots, d_n \in \{left, right\}$.

We assume that there is always an empty path from a node to itself.

$$x \in nodes(g) \Rightarrow path([], g, x, x)$$

A graph is called connected from a node x if for any node in the graph there is a path from x to that node.

$$connected(g, x) \triangleq \forall y \in nodes(g).\ \exists p.\ path(p, g, x, y)$$

where $nodes(g)$ is the set of nodes of graph g. The front of a set of nodes A in a graph is the set of nodes immediately reachable from nodes in A.

$$front(g, A) \triangleq \{x \mid \exists y \in nodes(g).\ x = y.left \vee x = y.right\}$$

A graph is maximal if the set of its nodes is a subset of the front of its nodes.

$$maximal(g) \triangleq nodes(g) \subseteq front(g, nodes(g))$$

A tree is a graph

$$istree(g, x) \triangleq \forall y \in nodes(g), \exists! p.\ path(p, g, x, y)$$

A graph g is strict subgraph of g' written as $g \subseteq_{strict} g'$ if the graph g can be obtained from g' by removing some nodes and children.

$$\begin{aligned} g \subseteq_{strict} g' \triangleq\ & \forall x \in nodes(g). \\ & (g.x.left = null \vee g.x.left = g'.x.left) \wedge \\ & (g.x.right = null \vee g.x.right = g'.x.right) \end{aligned}$$

A graph g is a spanning tree of a graph g' if

$$g \subseteq_{strict} g' \wedge istree(g, x) \wedge nodes(g) = nodes(g')$$

Lemma 1.

$$\forall A, front(g, A) \subseteq A \Rightarrow A = nodes(g)$$

1.1 The intuition

Let's assume `x` is a node of the graph g and the graph g is connected from x. Note that in such a case, any node `y` with a path from `x` to `y` is also a node of graph g. Therefore, in the recursive calls we can assume that whenever we have a call `span y`, `y` is a node of graph g.

Intuitively, `span x` returns true if it has managed to mark `x`. In that case, we know that there is a graph g' such that $g' \subseteq_{strict} g$, $istree(g', x)$, $maximal(g')$, nodes of g' are all marked and $front(g, nodes(g'))$ are marked (either by us through recursive calls or we have seen them marked through recursive calls).

If `span x` returns **`false`**, either `x` is **`null`** or we have seen it marked.

Let's confirm this intuition by going through the code. The call `span x` returns **`false`** if `x` is **`null`** (in line 3) or if the **`CAS`** of line 5 fails (in line 12). In the latter case, we know that `x.mark` is not **`false`** and there it is **`true`** (i.e., `x` is marked).

Otherwise, if the **`CAS`** of line 5 succeeds, the program returns **`true`** in line 10. In this case, we run `span` on the left and right child of `x` concurrently ad get result. In case `span` returns **`false`** for any of the children, we remove it. In what follows, we show that the desired properties hold.

First note that whenever we have a graph that consists of a node `x` and its children such that the children of `x` (if any) are maximal trees, the whole graph is a maximal tree. Furthermore, since the children (if any) are strict subgraphs of the original graph and for `x` we have only possibly removed children the whole graph is a strict subgraph of the original graph. Finally, note that front of the graph consisting of `x` and its children in the original graph is the front of the children of `x` in the original graph and their fronts. We know that these are all marked as guaranteed by the recursive calls on the children of `x` (they are either by the recursive calls or have been seen marked by the recursive calls).

At the top-level call to `span x`, we know that (as the pre-condition) `x` is not **`null`** and the graph we start with is not marked, it is maximal and it is it is connected from `x`. Therefore, the call to `span x` can't return **`false`** as it would indicate `x` was **`null`** or `x` was marked which is contradiction. Furthermore, we know that any marked node is marked (recursively) through this call.

Hence, at the top-level call to `span x` assuming `x` is a node of graph g, we know that there is a graph g' such that $g' \subseteq_{strict} g$, $istree(g', x)$, $maximal(g')$, nodes of g' are *exactly those nodes that are marked* and $front(g, nodes(g'))$ are all marked. Hence, we have $front(g, nodes(g')) \subseteq nodes(g')$. And by Lemma 1 we have $nodes(g) = nodes(g')$. Therefore, by definition of a spanning tree above we have g' is a spanning tree of g.

2. Proof in Iris

The main theorem proven in Irisi is `wp_span` below:

```
Lemma wp_span g markings (x : val) (l : loc) :
  l ∈ dom (gset _) g → maximal g → connected g l →
  heap_ctx ⋆
  ([ ⋆ map] l ↦ v ∈ g,
    ∃ (m : loc), markings !! l = Some m ⋆
      l ↦ (#m, children_to_val v) ⋆ m ↦ #false) ⊢
  WP span (SOME #l)
  {{ _, ∃ g',
        ([ ⋆ map] l ↦ v ∈ g',
          ∃ m : loc, markings !! l = Some m ⋆
            l ↦ (#m, children_to_val v) ⋆ m ↦ #true)
       ⋆ dom (gset _) g = dom (gset _) g'
       ⋆ ■ strict_subgraph g g' ⋆ ■ tree g' l }}.
```

The proof of correctness of `span` in Iris follows the intuitive reasoning above.

In Iris, we formalize graphs as finite maps from memory locations loc to pairs of memory $option\ loc$.

```
Definition graph := gmap loc (option loc ∗ option loc).
```