A Trees and Cursors

A.1 General Trees

We shall define a generic tree over an arbitrary type as being an element of that type coupled with a list of sub-trees, themselves trees of the same type

\[ T \in \text{Tree } A \cong A \times (\text{Tree } A)^* \]

This structure is very general and covers most concrete tree types, so long as the branching factor is finite. The tree picture here

![Tree Diagram](image)

can be represented by the following expression:

\[(a_1, (a_2, \Lambda, (a_3, ((a_4, \Lambda), (a_5, \Lambda))))\]

or alternatively:

\[
\begin{align*}
T_1 &= (a_1, (T_2, T_3)) \\
T_2 &= (a_2, \Lambda) \\
T_3 &= (a_3, (T_4, T_5)) \\
T_4 &= (a_4, \Lambda) \\
T_5 &= (a_5, \Lambda)
\end{align*}
\]

(Note that \(T_i\) is the tree whose root-node contains \(a_i\)).

We shall introduce the idea of sibling number \(\xi\) as being the index of a sub-tree in the list containing it and its siblings. The root of a tree always has its sibling number equal to one. The sibling numbers corresponding to the trees above are:

\[
\begin{align*}
\xi(T_1) &= 1 \\
\xi(T_2) &= 1 \\
\xi(T_3) &= 2 \\
\xi(T_4) &= 1 \\
\xi(T_5) &= 2
\end{align*}
\]

A.2 Abstract Syntax Trees

Abstract syntax trees are typically described recursively, using tagged disjoint unions of products (sum-of-products form). These can be represented using the general tree notion introduced above by introducing an invariant that requires each tree \((a, \tau)\) to be well-formed, where the component \(a\) now contains the
tag and other auxiliary information. The invariant will typically constrain the
length of \( \tau \), based on the value of \( a \).

Given a predicate \( P \) on Tree \( a \), we can introduce a tree invariant based on \( P \) as follows:

\[
\text{inv-Tree } a \quad : \quad (\text{Tree } a \to \mathbb{B}) \to \text{Tree } a \to \mathbb{B} \\
\text{inv-Tree}[P](a, \tau) \quad \triangleq \quad P(a, \tau) \land \forall [P]_{\tau}
\]

Consider a binary tree structure defined using sum-of-products form:

\[
\text{BTree } b \quad \triangleq \quad \text{LEAF } b \\
\quad \mid \quad \text{BRANCH } (\text{BTree } b) (\text{BTree } b)
\]

We can representing this using Tree by the following definitions:

\[
\text{BTreeNode } b \quad \triangleq \quad \text{LEAF } b \\
\quad \mid \quad \text{BRANCH } \\
\text{BTok} : \quad \text{Tree}(\text{BTreeNode } b) \to \mathbb{B} \\
\text{BTok}(\text{LEAF } \bot, \tau) \quad \triangleq \quad \text{1en } \tau = 0 \\
\text{BTok}(\text{BRANCH}, \tau) \quad \triangleq \quad \text{1en } \tau = 2 \\
\text{BTree } b \quad \triangleq \quad \text{Tree}(\text{BTreeNode } b) \mid \text{inv-Tree}[\text{BTok}]
\]

All that follows in this section can be applied to such trees by finding a mapping as just described. In general we shall work with standard abstract syntax tree definitions directly and apply the cursor operations to them directly, without making the translation explicit.

### A.3 Cursors

We wish to define the notion of a cursor which can point to a given (sub-)tree, and is capable of being moved around the tree:
The cursor needs not only to be able to identify the sub-tree to which it currently points, but also needs to be to find the parents of the sub-tree. We shall also want to easily identify the next sibling, of any given sub-tree. This leads us to view a cursor as (references to) a (sub-)tree along with a list of (references to) the parents in reverse order, each coupled with it position, or, in other words, a non empty list of (references to) trees and sibling numbers:

\[ c \in \text{Curs} A \equiv (N \times \text{Tree} A)^+ \]

We can identify the cursors above as:

\[ c_1 = \langle (1, T_2), (1, T_1) \rangle \]
\[ c_2 = \langle (1, T_1) \rangle \]
\[ c_3 = \langle (2, T_3), (1, T_1) \rangle \]
\[ c_4 = \langle (1, T_4), (2, T_3), (1, T_1) \rangle \]

We can view \( c_4 \) as:

![](image)

This structure may seem a little elaborate, but makes it very easy to define an invariant stating that a cursor must be a consistent match with a tree. By this is meant that any (sub-)tree entry in the cursor list is the designated sub-tree of the next entry:

\[ i_{\text{inv-Curs}} : \text{Curs} A \to \mathbb{B} \]
\[ i_{\text{inv-Curs}}(\langle s, T \rangle) \quad \triangleq \quad s = 1 \]
\[ i_{\text{inv-Curs}}(\langle s, T \rangle : (s', (a, \tau)) : c) \quad \triangleq \quad s \in \text{inds} \tau \]
\[ \land \quad T = \tau[s] \]
\[ \land \quad i_{\text{inv-Curs}}(\langle s', (a, \tau) \rangle : c) \]

**Recursion Termination Proof Obligation:**

Trivial - base case is singleton list, recursive call is applied to tail, and there is no empty-list case.

Another way of viewing the invariant is to consider the cursor as a sequence:

\[ \langle (s_1, (a_1, \tau_1)), \ldots, (s_i, (a_i, \tau_i)), \ldots, (s_n, (a_n, \tau_n)) \rangle \]

and the invariant as stating, for all \( i \in \{1 \ldots n - 1\} \), that:

\[ s_i \in \text{inds} \tau_{i+1} \quad \text{and} \quad t_i = \tau_{i+1}[s_i] \quad \text{and} \quad s_n = 1 \]

**A.4 Manipulating Cursors**

A cursor has no meaning independently of the tree with which it is associated, so we shall build and manipulate cursors by means of trees.
A.4.1 Make Cursor

Given a tree, we can define a cursor that points to the tree root node:

\[
\text{mkCurs} : \text{Tree A} \rightarrow \text{Curs A} \\
\text{mkCurs } T \triangleq \langle 1, T \rangle
\]

Invariant Preservation Proof Obligation:

\[\text{inv-Curs (mkCurs } T) \equiv \text{TRUE}\]

Proof:

\[
\begin{align*}
\text{inv-Curs (mkCurs } T) \\
= & \quad \text{(defn. of mkCurs)} \\
\text{inv-Curs } \langle 1, T \rangle \\
= & \quad \text{(1st case of inv-Curs)} \\
1 \equiv 1 \\
= & \quad \langle \rangle \\
\text{True}
\end{align*}
\]

A.4.2 Get Tree

To complete the circle, we provide a function to return the tree pointed to by a cursor (effectively discarding any ability to find parents):

\[
\text{getTree} : \text{Curs A} \rightarrow \text{Tree A} \\
\text{getTree } c \triangleq \pi_2(\text{hd } c)
\]

No Invariant Preservation Proof Obligation.
A.4.3 Down Cursor

Given a cursor, we can define a partial function that moves it down to the \( i \)th sub-tree, provided such a sub-tree exists:

\[
down C : \mathbb{N} \rightarrow \text{Curs} A \rightarrow \text{Curs} A
\]

\[
\text{pre-down} C[i] ((s, (a, \tau)) : c) \equiv i \in \text{inds} \tau
\]

\[
down C[i] ((s, (a, \tau)) : c) \equiv (i, \tau[i]) : (s, (a, \tau)) : c
\]

**Invariant Preservation Proof Obligation:**

\[
\text{inv-Curs} \ c \land \text{pre-down} C[i] c \Rightarrow i_{\text{inv-Curs}}(\down C[i]c)
\]

We shall assume \( \text{inv-Curs} \ c \) and \( \text{pre-down} C[i] c \), and seek to show \( i_{\text{inv-Curs}}(\down C[i]c) \).

We shall proceed by the cases:

\[
c = \langle (s, (a, \tau)) \rangle \quad \text{and} \quad c = \langle (s, (a, \tau)) : (s', (a', \tau')) : c \rangle
\]

**Case 1.** \( c = \langle (s, (a, \tau)) \rangle \) We assume:

\[
\text{inv-Curs} \langle (s, (a, \tau)) \rangle
\]

\[
\text{pre-down} C[i] \langle (s, (a, \tau)) \rangle : i \in \text{inds} \tau
\]

Proof of Case 1:

\[
\text{inv-Curs} (\down C[i] \langle (s, (a, \tau)) \rangle)
\]

\[
\text{(defn.} \ \down C) \quad \text{inv-Curs} \langle (i, \tau[i]), (s, (a, \tau)) \rangle
\]

\[
\text{(defn.} \ \text{inv-Curs}) \quad i \in \text{inds} \tau \land \tau[i] = \tau[i] \land \text{inv-Curs} \langle (s, (a, \tau)) \rangle
\]

\[
\text{(Ass.} \ \text{pre-down} C, \text{inv-Curs}) \quad \text{TRUE} \land \tau[i] = \tau[i] \land \text{TRUE}
\]

\[
\text{(def.of} \ \text{and} \ \text{prop.calc.)} \quad \text{TRUE}
\]

**Case 2.** \( c = \langle (s, (a, \tau)) : (s', (a', \tau')) : c \rangle \) We assume:

\[
\text{inv-Curs} \langle (s, (a, \tau)) : (s', (a', \tau')) : c \rangle
\]

\[
\text{pre-down} C[i] \langle (s, (a, \tau)) : (s', (a', \tau')) : c \rangle : i \in \text{inds} \tau
\]

Proof of Case 2:

\[
\text{inv-Curs} (\down C[i] \langle (s, (a, \tau)) : (s', (a', \tau')) : c \rangle)
\]

\[
\text{(defn.} \ \down C) \quad \text{inv-Curs} \langle (i, \tau[i]), (s, (a, \tau)) : (s', (a', \tau')) : c \rangle
\]

\[
\text{(defn.} \ \text{inv-Curs}) \quad i \in \text{inds} \land \tau[i] = \tau[i] \land \text{inv-Curs} \langle (s, (a, \tau)) : (s', (a', \tau')) : c \rangle
\]

\[
\text{(Ass.} \ \text{pre-down} C, \text{inv-Curs}) \quad \text{TRUE} \land \tau[i] = \tau[i] \land \text{TRUE}
\]

\[
\text{(def.of} \ \text{and} \ \text{prop.calc.)} \quad \text{TRUE}
\]

\[\blacksquare\]
A.4.4 Up Cursor

Given a cursor, we can define a partial function that moves it up to its parent tree, provided such a parent tree exists:

\[
\text{upC} : \text{Curs } A \rightarrow \text{Curs } A
\]
\[
\text{pre-upC } c \equiv \text{len } c > 1
\]
\[
\text{upC } c \equiv \text{tl } c
\]

**Invariant Preservation Proof Obligation:**

\[
\text{inv-Curs } c \land \text{pre-upC } c \Rightarrow \text{inv-Curs(} \text{upC } c \text{)}
\]

We shall assume \( \text{inv-Curs } c \) and \( \text{pre-upC } c \), and seek to show \( \text{inv-Curs(} \text{upC } c \text{)} \).

As the precondition only holds if \( \text{len } c > 1 \), we take the case

\[
c = (s, T) : (s', (a, \tau)) : c
\]

and assume the invariant:

\[
\text{inv-Curs}((s, T) : (s', (a, \tau)) : c) : s \in \text{inds } \tau \land T = \tau[s] \\
\land \text{inv-Curs}((s', (a, \tau)) : c)
\]

Proof:

\[
\text{inv-Curs(} \text{upC}((s, T) : (s', (a, \tau)) : c) \text{)}
\]
\[
= (\text{defn. upC})
\]
\[
\text{inv-Curs}((s', (a, \tau)) : c)
\]
\[
= (\text{Ass. inv-Curs})
\]
\[
\text{TRUE}
\]

\[\clubsuit\]
A.4.5 Next Cursor

Given a cursor, we can define a partial function that moves it to its next sibling tree, provided such a sibling tree exists:

\[
\text{nxtSib} : \text{Curs } A \rightarrow \text{Curs } A
\]

\[
\text{pre-nxtSib} ((s, T) : \Lambda) \equiv \text{FALSE}
\]

\[
\text{pre-nxtSib} ((s, T) : (s', (a, \tau)) : c) \equiv s + 1 \in \text{inds } \tau
\]

\[
\text{nxtSib} ((s, T) : (s', (a, \tau)) : c) \equiv (s + 1, \tau[s + 1] : (s', (a, \tau)) : c
\]

**Invariant Preservation Proof Obligation:**

\[
\text{inv-Curs } c \land \text{pre-nxtSib } c \Rightarrow \text{inv-Curs(nxtSib } c)
\]

We shall assume \text{inv-Curs } c and \text{pre-nxtSib } c, and seek to show \text{inv-Curs(nxtSib } c). As the precondition only holds if \text{len } c > 1, we take the case

\[
c = (s, T) : (s', (a, \tau)) : c'
\]

for arbitrary (possibly null) \(c'\), and assume precondition and invariant:

\[
\text{inv-Curs}((s, T) : (s', (a, \tau)) : c') : s \in \text{inds } \tau \land T = \tau[s]
\]

\[
\land \text{inv-Curs}(s', (a, \tau)) : c'
\]

\[
\text{pre-nxtSib}((s, T) : (s', (a, \tau)) : c') : s + 1 \in \text{inds } \tau
\]

**Proof:**

\[
\text{inv-Curs(nxtSib}((s, T) : (s', (a, \tau)) : c')
\]

\[
= (\text{defn. nxtSib})
\]

\[
\text{inv-Curs}((s + 1, \tau[s + 1]) : (s', (a, \tau)) : c')
\]

\[
= (\text{defn. inv-Curs})
\]

\[
s + 1 \in \text{inds } \tau \land \tau[s + 1] = \tau[s + 1] \land \text{inv-Curs}(s', (a, \tau)) : c'
\]

\[
= (\text{Ass. pre-nxtSib.inv-Curs})
\]

\[
\text{TRUE} \land \tau[s + 1] = \tau[s + 1] \land \text{TRUE}
\]

\[
= (\text{iff of } \land \text{ and prop. calc.})
\]

\[
\text{TRUE}
\]

A.4.6 Cursor Children Number

We often want to know the number of children associated with the current cursor:

\[
\text{chldNum} : \text{Curs } A \rightarrow \mathbb{N}
\]

\[
\text{chldNum } ((\_, (\_ : \tau)) : \_ ) \equiv \text{len } \tau
\]

A.4.7 Cursor Sibling Number

We often want to know the current sibling number:

\[
\text{sibNo} : \text{Curs } A \rightarrow \mathbb{N}
\]

\[
\text{sibNo } ((s, \_ : \_ ) \equiv s
\]
A.4.8 Root Cursor

We might want to know when a cursor denotes the tree root:

\[
\text{isRoot} : Curs \ A \rightarrow \mathbb{B} \\
\text{isRoot } c \triangleq c \equiv \langle 1, \_ \rangle
\]

A.4.9 Leaf Cursor

We might want to know when a cursor denotes a leaf node:

\[
\text{isLeaf} : Curs \ A \rightarrow \mathbb{B} \\
\text{isLeaf } c \triangleq (\pi_2 \circ \pi_2 \circ \text{hd})|c = \Lambda
\]
A.4.10 First Leaf Cursor

Given a cursor, we can define a partial function that moves it to the first leaf in the (sub-)tree denoted by that cursor:

\[
\text{fstLeaf} : \ Curs \ A \rightarrow \ Curs \ A
\]

\[
\text{fstLeaf} \ c \ \triangleright\ \text{isLeaf} \ c \rightarrow \ c,
\]

\[
\rightarrow \text{fstLeaf(downC}[1]c)\]

Recursion Termination Proof Obligation:

Let \( h(c) \) denote the height of the (sub-)tree denoted by \( c \). Then, \( \text{isLeaf} \ c \equiv h(c) = 1 \) and \( \text{downC} \) reduces \( h \) by 1.

Invariant Preservation Proof Obligation:

\[
i_{\text{inv-Curs}} \ c \Rightarrow i_{\text{inv-Curs}}(\text{fstLeaf} \ c)
\]

We shall prove this by induction on the height of \( c \):

\[
P(n) \equiv h(c) = n \Rightarrow (\text{inv-Curs} \ c \Rightarrow i_{\text{inv-Curs}}(\text{fstLeaf} \ c))
\]

Base Case: \( h(c) = 1 \)

\[
h(c) = 1 \Rightarrow (\text{inv-Curs} \ c \Rightarrow i_{\text{inv-Curs}}(\text{fstLeaf} \ c))
\]

We shall assume \( h(c) = 1 \)

\[
i_{\text{inv-Curs}} \ c \Rightarrow i_{\text{inv-Curs}}(\text{fstLeaf} \ c)
\]

\[
= \ (\text{defn. } \text{fstLeaf})
\]

\[
i_{\text{inv-Curs}} \ c \Rightarrow i_{\text{inv-Curs}}(\text{isLeaf} \ c \rightarrow \ c, \ldots)
\]

\[
= \ (\text{prop. } h(c) = 1 \equiv \text{isLeaf} \ c, \text{ conditional})
\]

\[
i_{\text{inv-Curs}} \ c \Rightarrow i_{\text{inv-Curs}} \ c
\]

\[
= \ (\text{prop. calc.})
\]

\[
\text{TRUE}
\]

Inductive step: \( h(c) = n \)

we assume this property holds for all \( c' \) where \( h(c') < n \), for \( n > 1 \). We show that it will hold for \( c \) with height \( n \). We assume:

\[
h(c') < n \Rightarrow (\text{inv-Curs} \ c' \Rightarrow i_{\text{inv-Curs}}(\text{fstLeaf} \ c'))
\]

and

\[
h(c) = n
\]

We want to show:

\[
i_{\text{inv-Curs}} \ c \Rightarrow i_{\text{inv-Curs}}(\text{fstLeaf} \ c)
\]

We begin by stating a previous proof result, stating that \( \text{downC} \) preserves the invariant, justified by the fact that the precondition is satisfied \( (h(c) > 1 \Rightarrow \text{pre-downC}[1]c) \) and our assumption, instantiated with \( c' = \text{downC}[1]c \), which is justified by the property that \( h(\text{downC}[1]c) < h(c) \):
\[\text{inv-Curs}(c) \Rightarrow \text{inv-Curs(downC[1]c)}\]
\[\wedge\]
\[\text{inv-Curs(downC[1]c)} \Rightarrow \text{inv-Curs(fstLeaf(downC[1]c)})\]
\[\Rightarrow (A \Rightarrow B \wedge B \Rightarrow C \Rightarrow A \Rightarrow C)\]
\[\text{inv-Curs(c)} \Rightarrow \text{inv-Curs(fstLeaf(downC[1]c))}\]
\[= \text{(prop. } h(c) > 1 \equiv \neg \text{Leaf c, conditional)}\]
\[\text{inv-Curs c} \Rightarrow \text{inv-Curs}(\text{isLeaf c} \Rightarrow c, \text{fstLeaf(downC[1]c)})\]
\[= \text{(defn. } \text{fstLeaf})\]
\[\text{inv-Curs c} \Rightarrow \text{inv-Curs(fstLeaf c)}\]

### A.4.11 Last Cursor

We might want to know when a cursor lies along the last sibling edges in the tree. In the event that the cursor is a leaf cursor, then it denotes the last leaf.

\[
\begin{align*}
\text{isLast} & : \text{Cars A} \rightarrow B \\
\text{isLast c} & \equiv \text{isRoot c} \rightarrow \text{TRUE}, \\
& \ \text{sibNo c = chldNum(upC c)} \\
& \ \wedge \ \text{isLast(upC c)}
\end{align*}
\]

**Recursion Termination Proof Obligation:** The base case is when the cursor is root, i.e. a singleton list. The recursive call is to upC, which shortens the cursor list by one.
A.4.12 Find First non-Last Ancestor Cursor

Given a cursor, we define a function that searches upward for the first ancestor which is not itself a “last” sibling:

\[
\text{non\text{-}LAnC} : Curs A \rightarrow Curs A
\]

\[
\text{pre\text{-}non\text{-}LAnC} \ c \ \equiv \ \neg\text{is\text{-}Last} \ c
\]

\[
\text{non\text{-}LAnC} \ c \ \equiv \ \begin{cases} \text{if sibNo} \ c < \text{childNum}(\text{upC} \ c) \\ \text{then} \ c \\ \text{else} \ \text{non\text{-}LAnC}(\text{upC} \ c) \end{cases}
\]

Recursion Termination Proof Obligation:
The base case is a non-root cursor whose head meets a certain condition (Note that \(\text{isRoot} \ c \Rightarrow \text{isLast} \ c\)). The recursive case shortens the cursor length. The pre-condition ensures that the base case is reached before the cursor becomes singleton, i.e. root.

Invariant Preservation Proof Obligation:

\[
\text{inv}\text{-}Curs \ c \land \text{pre\text{-}non\text{-}LAnC} \ c \ \Rightarrow \ \text{inv}\text{-}Curs(\text{non\text{-}LAnC} \ c)
\]

We first observe that the precondition requires \(\text{len} \ c > 1\), so we shall instantiate \(c\) as follows:

\[
c = c_1
\]

\[
c_1 = (s_1, T_1) : c_2
\]

\[
c_2 = (s_2, T_2) : c'
\]

\[
T_i = (a_i, \tau_i), \quad i \in \{1, 2\}
\]

where \(c'\) is the rest of the cursor (possibly null). Note that the following identities hold:

\[
s_i = \text{sibNo} \ c_i
\]

\[
\text{childNum} \ c_i = \text{len} \ \tau_i
\]

\[
c_2 = \text{upC} \ c_1
\]

By expanding all definitions using the instantiation above, and distributing a function application through a conditional, we are being asked to prove that assuming the invariant,

\[
s_1 \in \text{inds} \ \tau_2 \land T_1 = \tau_2[s_1] \land \text{inv}\text{-}Curs \ c_2
\]

and precondition,

\[
\neg(s_1 = \text{len} \ \tau_2 \land \text{is\text{-}Last} \ c_2)
\]

leads to the conclusion

\[
s_1 < \text{len} \ \tau_2 \rightarrow \text{inv}\text{-}Curs \ c_1, \ \text{inv}\text{-}Curs(\text{non\text{-}LAnC} \ c_2)
\]

We proceed by case-analysis, noting that \(s_1\) is either less than or equal to \(\text{len} \ \tau_2\) (it can’t be greater, by the invariant).

Case 1: \(s_1 < \text{len} \ \tau_2\)
\[ s_1 < \text{len } \tau_2 \rightarrow \text{inv-Cars } c_2, \text{ inv-Cars}(\text{nonLAnC } c_2) \]

= (Case 1, conditional)

\[ \text{inv-Cars } c_1 \]

= (Ass. (inv))

\[ True \]

Case 2: \( s_1 = \text{len } \tau_2 \)

\[ s_1 < \text{len } \tau_2 \rightarrow \text{inv-Cars } c_1, \text{ inv-Cars}(\text{nonLAnC } c_2) \]

= (Case 2, conditional)

\[ \text{inv-Cars}(\text{nonLAnC } c_2) \]

We need to show this subject to the assumptions given above, i.e.
We shall prove this by induction on the length of \( c \).

Base Case: \( \text{len } c = 2 \)

In this case we have \( c_2 = \langle (s_2, T_2) \rangle \) so, after expansion of \( \text{inv-Cars } c_2 \), our proof obligation becomes,

\[ s_1 = \text{len } \tau_2 \land s_1 \in \text{inds } \tau_2 \land T_1 = \tau_2[s_1] \]

\[ \land s_2 = 1 \]

\[ \land \neg(s_1 = \text{len } \tau_2 \land \text{isLast}(\langle s_2, T_2 \rangle)) \]

\[ \Rightarrow \text{inv-Cars}(\text{nonLAnC } c_2) \]

However, \( \text{isLast}(\langle 1, T_2 \rangle) \equiv True \), so the antecedent is false, meaning the implication is true.

Inductive Step: \( \text{len } c > 2 \)

We shall assume the law true for \( c_1 \),

\[ s_1 = \text{len } \tau_2 \land s_1 \in \text{inds } \tau_2 \land T_1 = \tau_2[s_1] \]

\[ \land \text{inv-Cars } c_2 \]

\[ \land \neg(s_1 = \text{len } \tau_2 \land \text{isLast } c_2) \]

\[ \Rightarrow \text{inv-Cars}(\text{nonLAnC } c_2) \]

and seek to show that this implies it is true for \( c_0 \), which is defined as

\[ c_0 = \langle s_0, T_0 \rangle : c_1 \quad \text{where } \quad T_0 = (a_0, \tau_0) \]

i.e.

\[ s_0 = \text{len } \tau_1 \land s_0 \in \text{inds } \tau_1 \land T_0 = \tau_1[s_0] \]

\[ \land \text{inv-Cars } c_1 \]

\[ \land \neg(s_0 = \text{len } \tau_1 \land \text{isLast } c_1) \]

\[ \Rightarrow \text{inv-Cars}(\text{nonLAnC } c_1) \]

We expand the consequent and distribute the invariant through the conditional:

\[ s_0 = \text{len } \tau_1 \land s_0 \in \text{inds } \tau_1 \land T_0 = \tau_1[s_0] \]

\[ \land \text{inv-Cars } c_1 \]

\[ \land \neg(s_0 = \text{len } \tau_1 \land \text{isLast } c_1) \]

\[ \Rightarrow s_1 < \text{len } \tau_2 \rightarrow \text{inv-Cars } c_1, \text{ inv-Cars}(\text{nonLAnC } c_2) \]
If $s_1 < \text{len } \tau_2$ then we have $\text{inv-Curs } c_1$ in both antecedent and consequent, so it is trivially true.

If $s_1 = \text{len } \tau_2$ then we want to show:

\[
\begin{align*}
\sigma_0 &= \text{len } \tau_1 \land \sigma_0 \in \text{inds } \tau_1 \land T_0 = \tau_1[\sigma_0] \\
\land & s_1 = \text{len } \tau_2 \\
\land & \text{inv-Curs } c_1 \\
\land & \neg(s_0 = \text{len } \tau_1 \land \text{isLast } c_1) \\
\Rightarrow & \text{inv-Curs}(\text{nonLAnc } c_2)
\end{align*}
\]

We can simplify the last antecedent using the first:

\[
\begin{align*}
\sigma_0 &= \text{len } \tau_1 \land \sigma_0 \in \text{inds } \tau_1 \land T_0 = \tau_1[\sigma_0] \\
\land & s_1 = \text{len } \tau_2 \\
\land & \text{inv-Curs } c_1 \\
\land & \neg\text{isLast } c_1 \\
\Rightarrow & \text{inv-Curs}(\text{nonLAnc } c_2)
\end{align*}
\]

We expand isLast $c_1$, using fact $\neg\text{isRoot } c_1$

\[
\begin{align*}
\sigma_0 &= \text{len } \tau_1 \land \sigma_0 \in \text{inds } \tau_1 \land T_0 = \tau_1[\sigma_0] \\
\land & s_1 = \text{len } \tau_2 \\
\land & \text{inv-Curs } c_1 \\
\land & \neg(s_1 = \text{len } \tau_2 \land \text{isLast } c_2) \\
\Rightarrow & \text{inv-Curs}(\text{nonLAnc } c_2)
\end{align*}
\]

we use the 4th antecedent to simplify last:

\[
\begin{align*}
\sigma_0 &= \text{len } \tau_1 \land \sigma_0 \in \text{inds } \tau_1 \land T_0 = \tau_1[\sigma_0] \\
\land & s_1 = \text{len } \tau_2 \\
\land & \text{inv-Curs } c_1 \\
\land & \neg\text{isLast } c_2 \\
\Rightarrow & \text{inv-Curs}(\text{nonLAnc } c_2)
\end{align*}
\]

Expanding $\text{inv-Curs } c_1$ gives antecedents that include the hypothesis antecedents, so we can then assert its consequent, which is what we are try to show here. ✷
A.4.13 Next Leaf Cursor

Given a leaf cursor, we can define a partial function that moves it to the next leaf in the tree, provided such a leaf exists. We in fact handle the case of a general (non-root) cursor, in which case we return the first leaf of its next sibling

\[
\text{nxtLeaf} : \text{Curs } A \rightarrow \text{Curs } A
\]
\[
\text{pre-nxtLeaf } \text{c} \ \equiv \ \neg \text{isLast } \text{c}
\]
\[
\text{nxtLeaf } \text{c} \ \equiv \ \text{fstLeaf} \circ \text{nxtSib} \circ \text{nonLAnc} \text{c}
\]

Invariant Preservation Proof Obligation:

\[
\begin{align*}
\text{inv-Curs } \text{c} \\
& \land \ \neg \text{isLast } \text{c} \\
& \Rightarrow \ \text{pre-nxtSib} (\text{nxtLeaf } \text{c})
\end{align*}
\]

The precondition for \text{nxtLeaf} is identical to that for \text{nonLAnc}, being \neg \text{isLast } \text{c}, so we can deduce that the call to it will preserve the invariant. The operation \text{fstLeaf} is total and preserves the invariant. So all we need to show is that the result of \text{nonLAnc} satisfies the precondition of \text{nxtSib}, given that the call to \text{nonLAnc} is itself OK:

\[
\begin{align*}
\text{inv-Curs } \text{c} \\
& \land \ \neg \text{isLast } \text{c} \\
& \Rightarrow \ \text{pre-nxtSib} (\text{nonLAnc } \text{c})
\end{align*}
\]

First we note that \text{pre-nxtSib} (\text{c}) implies that \text{c} is \text{non-Root}, i.e. that \text{len } \text{c} > 1.

We shall instantiate \text{c} as follows:

\[
\begin{align*}
\text{c} &= \text{c}_1 \\
\text{c}_1 &= (s_1, T_1) : \text{c}_2 \\
\text{c}_2 &= (s_2, T_2) : c' \\
T_i &= (a_i, \tau_i), \ i \in \{1, 2\}
\end{align*}
\]

where \(c'\) is the rest of the cursor (possibly null). Instantiating up to \text{c}_1 we obtain:

\[
\begin{align*}
\text{inv-Curs} ((s_1, T_1) : \text{c}_2) \\
& \land \ \neg \text{isLast} ((s_1, T_1) : \text{c}_2) \\
& \Rightarrow \ \text{pre-nxtSib} (\text{nonLAnc} ((s_1, T_1) : \text{c}_2))
\end{align*}
\]

We expand definitions, noting that \text{isRoot } \text{c} = \text{FALSE} in passing:

\[
\begin{align*}
s_1 \in \text{inds } \tau_2 & \land T_1 = \tau_2 [s_1] \land \text{inv-Curs } \text{c}_2 \\
& \land \ \neg (s_1 = \text{len } \tau_2 \land \text{isLast } \text{c}_2) \\
& \Rightarrow \ \text{pre-nxtSib} (s_1 < \text{len } \tau_2 \rightarrow ((s_1, T_1) : \text{c}_2) \circ (\text{nonLAnc } \text{c}_2))
\end{align*}
\]

We perform a case split:

Case 1: \(s_1 < \text{len } \tau_2\)
We add the case assumption, simplify and evaluate the conditional:

\[ s_1 < \text{len } \tau_2 \land T_1 = \tau_2[s_1] \land \text{inv-Curs } c_2 \]
\[ \land \neg(s_1 = \text{len } \tau_2 \land \text{isLast } c_2) \]
\[ \Rightarrow \text{pre-nxtSib}((s_1, T_1); c_2) \]

We expand defn. of pre-nxtSib:

\[ s_1 \in \text{inds } \tau_2 \land T_1 = \tau_2[s_1] \land \text{inv-Curs } c_2 \]
\[ \land \neg(s_1 = \text{len } \tau_2 \land \text{isLast } c_2) \]
\[ \Rightarrow s_1 + 1 \in \text{inds } \tau_2 \]

But, \( s_1 < \text{len } \tau_2 \Rightarrow s_1 + 1 \in \text{inds } \tau_2 \), so we are done.

Case 2: \( s_1 = \text{len } \tau_2 \)

We add the case assumption, simplify and evaluate the conditional:

\[ s_1 = \text{len } \tau_2 \land T_1 = \tau_2[s_1] \land \text{inv-Curs } c_2 \]
\[ \land \neg(s_1 = \text{len } \tau_2 \land \text{isLast } c_2) \]
\[ \Rightarrow \text{pre-nxtSib}((\text{nonLAnc } c_2)) \]

We shall prove this by induction on the length of \( c \).

Base Case: \( \text{len } c = 2 \)

Expanding \( c_2 \) with \( c' = \Lambda \):

\[ s_1 = \text{len } \tau_2 \land T_1 = \tau_2[s_1] \land \text{inv-Curs}((s_2, T_2)) \]
\[ \land \neg(\text{isRoot}((s_2, T_2))) \]
\[ \Rightarrow \text{pre-nxtSib}((\text{nonLAnc}((s_2, T_2)))) \]

Expanding definition of \( \text{inv-Curs} \) and isLast:

\[ s_1 = \text{len } \tau_2 \land T_1 = \tau_2[s_1] \land s_2 = 1 \]
\[ \land \neg(\text{isRoot}((s_2, T_2))) \Rightarrow \text{TRUE, \ldots} \]
\[ \Rightarrow \text{pre-nxtSib}((\text{nonLAnc}((s_2, T_2)))) \]

Expanding definition of isRoot:

\[ s_1 = \text{len } \tau_2 \land T_1 = \tau_2[s_1] \land s_2 = 1 \]
\[ \land \neg(s_2 = 1 \Rightarrow \text{TRUE, \ldots}) \]
\[ \Rightarrow \text{pre-nxtSib}((\text{nonLAnc}((s_2, T_2)))) \]

The antecedent is false, so the implication immediately follows.

Inductive Case: \( \text{len } c > 2 \)
We assume
\[ s_1 = \text{len} \, \tau_2 \land T_1 = \tau_2[s_1] \land \text{inv-Curs } c_2 \]
\[ \land \neg \text{isLast } c_2 \]
\[ \Rightarrow \text{pre-nxtSib}(\text{nonLAnC } c_2) \]
and show it implies
\[ s_0 = \text{len} \, \tau_1 \land T_0 = \tau_1[s_0] \land \text{inv-Curs } c_1 \]
\[ \land \neg \text{isLast } c_1 \]
\[ \Rightarrow \text{pre-nxtSib}(\text{nonLAnC } c_1) \]
where \( c_0 = (s_0, T_0) : c_1 \).

We expand the consequent and distribute the pre-condition through the conditional:
\[ s_0 = \text{len} \, \tau_1 \land T_0 = \tau_1[s_0] \land \text{inv-Curs } c_1 \]
\[ \land \neg \text{isLast } c_1 \land s_1 < \text{len} \, \tau_2 \]
\[ \Rightarrow s_1 < \text{len} \, \tau_2 \rightarrow \text{pre-nxtSib } c_1 \land \text{pre-nxtSib}(\text{nonLAnC } c_2) \]
We do a case split
Case 1: \( s_1 < \text{len} \, \tau_2 \)
Adding the case, and evaluating the conditional
\[ s_0 = \text{len} \, \tau_1 \land T_0 = \tau_1[s_0] \land \text{inv-Curs } c_1 \]
\[ \land \neg \text{isLast } c_1 \land s_1 < \text{len} \, \tau_2 \]
\[ \Rightarrow \text{pre-nxtSib } c_1 \]
Evaluating pre-nxtSib, noting that \( \neg \text{Root } c_1 \) is true:
\[ s_0 = \text{len} \, \tau_1 \land T_0 = \tau_1[s_0] \land \text{inv-Curs } c_1 \]
\[ \land \neg \text{isLast } c_1 \land s_1 < \text{len} \, \tau_2 \]
\[ \Rightarrow s_1 + 1 \in \text{inds } \tau_2 \]
This reduces to true, given that \( s_1 + 1 \in \text{inds } \tau_2 \equiv s_1 < \text{len } \tau \).
Case 2: \( s_1 = \text{len} \, \tau_2 \)
Adding the case, and evaluating the conditional
\[ s_0 = \text{len} \, \tau_1 \land T_0 = \tau_1[s_0] \land \text{inv-Curs } c_1 \]
\[ \land \neg \text{isLast } c_1 \land s_1 = \text{len} \, \tau_2 \]
\[ \Rightarrow \text{pre-nxtSib}(\text{nonLAnC } c_2) \]
Expand isLast \( c_1 \)
\[ s_0 = \text{len} \, \tau_1 \land T_0 = \tau_1[s_0] \land \text{inv-Curs } c_1 \]
\[ \land \neg (s_1 = \text{len} \, \tau_2 \land \text{isLast } c_2) \land s_1 = \text{len} \, \tau_2 \]
\[ \Rightarrow \text{pre-nxtSib}(\text{nonLAnC } c_2) \]
Use case to simplify:

\[ s_0 = \text{len} \tau_1 \land T_0 = \tau_1[s_0] \land \text{inv-Curs } c_1 \land \neg \text{isLast } c_2 \land s_1 = \text{len} \tau_2 \Rightarrow \text{pre-nxtSib(nonLAnc } c_2) \]

Expand \textit{inv-Curs } c_1:

\[ s_0 = \text{len} \tau_1 \land T_0 = \tau_1[s_0] \land s_1 \in \text{inds } \tau_1 \land T_1 = \tau_2[s_1] \land \text{inv-Curs } c_2 \land \neg \text{isLast } c_2 \land s_1 = \text{len} \tau_2 \Rightarrow \text{pre-nxtSib(nonLAnc } c_2) \]

We now have all the hypothesis antecedents present, so the consequent follows.

\[ \blacklozenge \]

A.4.14 All Children Cursors

Given a leaf cursor, we can define a function returns a sequence of cursors, each denoting a child:

\[
\begin{align*}
\text{allChild} : & \quad \text{Curs A} \rightarrow (\text{Curs A})^* \\
\text{allChild } c & \equiv \text{down}^*(1\ldots\text{siNo } c) \\
\text{where} \\
\text{down } i & \equiv \text{downC}[i]c
\end{align*}
\]

Invariant Preservation Proof Obligation:

\[
\text{inv-Curs } c \Rightarrow \forall[\text{inv-Curs}](\text{allChild } c)
\]

We observe that we simply build a sequence by evaluating \text{downC}[i]c for every \( i \in \text{inds } c \). Each call satisfies the pre-condition, and this operator has already been shown to preserve the invariant. If there are no children then an empty list is returned which vacuously satisfies the \( \forall \).

\[ \blacklozenge \]
A.5 Shorthands

We introduce some shorthands to produce some more terse expressions.

\[
\begin{align*}
\text{Curs } T & \equiv \text{mkCurs } T \\
\text{Tree } c & \equiv \text{getTree } c \\
\text{c ↓ i} & \equiv \text{downC[i]} c \\
\text{c ↑} & \equiv \text{upC } c \\
\text{c ↑} & \equiv \text{nxtSib } c \\
\#c & \equiv \text{childNum } c \\
\&c & \equiv \text{sibNo } c \\
\text{R } c & \equiv \text{isRoot } c \\
\text{L } c & \equiv \text{isLeaf } c \\
\alpha c & \equiv \text{fstLeaf } c \\
\Omega c & \equiv \text{isLast } c \\
\&c & \equiv \text{nonLAnc } c \\
\text{c ⊥} & \equiv \text{nxtLeaf } c \\
\text{C } c & \equiv \text{allChld } c
\end{align*}
\]
A.6 Shorthand Definitions

We re-state all the definitions here using these shorthands.

\[
\begin{align*}
Curs & : \quad Tree A \rightarrow Curs A \\
Curs T & \equiv \langle (1, T) \rangle \\
Tree & : \quad Curs A \rightarrow Tree A \\
Tree c & \equiv \pi_2(\text{hd } c) \\
\downarrow i & : \quad Curs A \rightarrow Curs A \\
& \quad ! \quad i \in \text{inds } \tau \\
((s, (a, \tau)) : c) \downarrow i & \equiv (i, \tau[i]) : (s, (a, \tau)) : c \\
\uparrow & : \quad Curs A \rightarrow Curs A \\
& \quad ! \quad \text{len } c > 1 \\
\mathbf{c} \uparrow & \equiv \text{tl } \mathbf{c} \\
\uparrow & : \quad Curs A \rightarrow Curs A \\
((s, T) : A) & \quad ! \quad \text{false} \\
& \quad ! \quad s + 1 \in \text{inds } \tau \\
((s, T) : (s', a, \tau)) : c \uparrow & \equiv (s + 1, \tau[s + 1]) : (s', (a, \tau)) : c \\
\# & : \quad Curs A \rightarrow \mathbb{N} \\
\#((s, \tau)) : \downarrow & \equiv \text{len } \tau \\
\downarrow & : \quad Curs A \rightarrow \mathbb{N} \\
\downarrow((s, \tau)) : \downarrow & \equiv s
\end{align*}
\]
\[
\begin{align*}
R & : \text{Curs } A \to \mathbb{B} \\
R_e & \triangleq e \equiv \langle 1, \omega \rangle \\
L & : \text{Curs } A \to \mathbb{B} \\
L_e & \triangleq (\pi_2 \circ \pi_2 \circ \text{hd})e = \Lambda \\
\alpha & : \text{Curs } A \to \text{Curs } A \\
\alpha_e & \triangleq L_e \rightarrow e, \\
& \quad \rightarrow \alpha(e \downarrow 1) \\
\Omega & : \text{Curs } A \to \mathbb{B} \\
\Omega_e & \triangleq R_e \rightarrow \text{TRUE}, \\
& \quad \rightarrow \#(e) \land \Omega(e) \\
\exists & : \text{Curs } A \to \text{Curs } A \\
\exists_e & \triangleq \neg R_e \land \neg \Omega_e \\
\exists_e & \triangleq \#(e) < \#(e) \rightarrow e, \quad \exists_e \\
\downarrow & : \text{Curs } A \to \text{Curs } A \\
\downarrow & \triangleq \neg R_e \land \neg \Omega_e \\
\downarrow_e & \triangleq \alpha((\exists_e) \uparrow) \\
\mathcal{C} & : \text{Curs } A \to \text{Curs } A \\
\mathcal{C}_e & \triangleq D^s(1 \ldots \#e) \text{ where } D_i \equiv e \downarrow i
\end{align*}
\]
A.7 Height Properties

In order to discharge some proof obligations we need to perform induction over tree heights. Here we develop a small sub-theory linking tree heights to cursor lengths and other related attributes.

We define the height of a tree as follows:

\[ h : \text{Tree} \rightarrow \mathbb{N} \]
\[ h(a, \tau) \triangleq 1 + \max(h^a\tau) \]
\[ \text{where } \max\Lambda = 0 \]

The height of a cursor is the height of the (sub-)tree indicated by it:

\[ h : \text{Curs} \rightarrow \mathbb{N} \]
\[ h((s, T) : e) \triangleq h(T) \]

Note that we overload the symbol \( h \) — its use will be clear from context.

Property Summary

\[ h(e) \geq 1 \]
\[ h(e \downarrow i) < h(e) \]
\[ h(e) < h(e \uparrow) \]
\[ L e \equiv h(e) = 1 \]

In all cases above, we have side conditions stating that all invariants and pre-conditions hold. These are made explicit in the proofs.
A.7.1 Proof of $h(c) \geq 1$

\[
    h((s, (a, \tau)) : c') =
    \begin{align*}
        \text{(defn. $h$ on $C_{ars}$)} \\
        h(a, \tau) \\
        \geq \text{(arithmetic)} \\
        1
    \end{align*}
\]

A.7.2 Proof of $h(c \downarrow i) < h(c)$

Let $c = (s, (a, \tau)) : c'$. The precondition requires that \( i \in \text{inds } \tau \).

\[
    h((s, (a, \tau)) : c') =
    \begin{align*}
        \text{(defn. $h$ on $C_{ars}$)} \\
        h(a, \tau) \\
        \geq \text{(arithmetic)} \\
        \max(h^*\tau) \\
        > \text{(prop. of max)} \\
        h(\tau[i]) \\
        = \text{(defn of $h$ on $C_{ars}$)} \\
        h((i, \tau[i]) : (s, (a, \tau)) : c') \\
        = \text{(defn of $\downarrow$)} \\
        h(((s, (a, \tau)) : c') \downarrow i)
    \end{align*}
\]
A.7.3 Proof of $h(c) < h(c \uparrow)$

Let $c = (s, (a, \tau)) : c'$. Precondition requires $1 \leq c > 1$, so let $c = (s_1, \tau_2[s_1]) : (s_2, (a_2, \tau_2)) : c'$.

\[
\begin{align*}
    h((s_1, \tau_2[s_1]) : (s_2, (a_2, \tau_2)) : c') &= (\text{defn. } h \text{ on } \textit{Curs}) \\hline
    h(\tau_2[s_1]) \leq (\text{prop. of max}) \\hline
    \text{max}(h^*\tau_2) < (\text{arithmetic}) \\hline
    1 + \text{max}(h^*\tau_2) &= (\text{defn of } h \text{ on } \textit{Tree}) \\hline
    h(a_2, \tau_2) &= (\text{defn. of } h \text{ on } \textit{Curs}) \\hline
    h((s_2, (a_2, \tau_2)) : c') &= (\text{defn. of } \uparrow) \\hline
    h(((s_1, \tau_2[s_1]) : (s_2, (a_2, \tau_2)) : c') \uparrow)
\end{align*}
\]
A.7.4 Proof of $Lc \equiv h(c) = 1$

Let $c = (s, (a, \tau)) : c'$.

Case 1: $Lc \Rightarrow h(c) = 1$

\[
L((s, (a, \tau)) : c') = (\text{defn. } L) \\
(\pi_2 \circ \pi_2 \circ \text{hd}) ((s, (a, \tau)) : c') = \Lambda \\
= (\text{defn or } \pi_2 \text{ and } \text{hd}) \\
\tau = \Lambda \\
\Rightarrow (\text{rewriting } \tau) \\
h(c) = h((s, (a, \Lambda)) : c')) = (\text{def. of } h \text{ on } Cur)$ \\
h(c) = h(a, \Lambda) = (\text{def. of } h \text{ on } Tree) \\
h(c) = 1 + \max(h^*\Lambda) = (\text{prop. of } \max) \\
h(c) = 1 + 0 = (\text{arithmetic}) \\
h(c) = 1
\]

Case 2: $Lc \Leftarrow h(c) = 1$

\[
h((s, (a, \tau)) : c') = 1 = (\text{def. } h \text{ on } Cur) \\
h(a, \tau) = 1 = (\text{def. } h \text{ on } Tree) \\
1 + \max(h^*\tau) = 1 = (\text{arithmetic}) \\
\max(h^*\tau) = 0 = (\text{prop. of } \max, \text{ given that } h(c) > 0 \text{ for all } c) \\
\tau = \Lambda = (\text{defn or } \pi_2 \text{ and } \text{hd}) \\
(\pi_2 \circ \pi_2 \circ \text{hd}) ((s, (a, \tau)) : c') = \Lambda = (\text{defn. } L) \\
L((s, (a, \tau)) : c')
\]

\[\text{\bullet}\]
A.8 Cursor Properties

Property Summary

\[ \text{Tree}(\text{Curs } T) = T \]
\[ (\mathbf{c} \downarrow i) \uparrow = \mathbf{c} \]
\[ (\lfloor \mathbf{c} \rfloor) \in \{1 \ldots \#(\mathbf{c} \uparrow)\} \]
\[ (\mathbf{c} \uparrow) \downarrow (\lfloor \mathbf{c} \rfloor) = \mathbf{c} \]
\[ \mathbf{c} \downarrow = (\mathbf{c} \uparrow) \downarrow (\lfloor \mathbf{c} \rfloor + 1) \]
\[ (\mathbf{c} \downarrow) \uparrow = \mathbf{c} \uparrow \]
\[ L(\alpha \mathbf{c}) = \text{TRUE} \]
\[ L(\mathbf{c} \parallel) = \text{TRUE} \]
\[ R \mathbf{c} \Rightarrow \Omega \mathbf{c} \]
\[ R(\text{Curs}(\text{Tree } \mathbf{c})) = \text{TRUE} \]

In all cases above, we have side conditions stating that all invariants and preconditions hold. These are made explicit in the proofs.

In particular, if \text{len } \mathbf{c} > 1, then the cursor will have form \( \mathbf{c}_1 : \mathbf{c}_2 : \mathbf{c}' \) where \( \mathbf{c}_i = (s_i, (a_i, \tau_i)) \). However the invariant requires \((a_1, \tau_1) = \tau_2[s_1] \), so when we instantiate \( \mathbf{c} \) of length greater than one, we do so as

\[ \mathbf{c} = (s_1 \tau_2[s_1]) : (s_2, (a_2, \tau_2)) : \mathbf{c}' \]
A.8.1 Proof of $\text{Tree}(\text{Curs } T) = T$

$$\begin{align*}
\text{Tree}(\text{Curs } T) \\
= & \text{(def. } \text{Curs)} \\
= & \text{(def. } \text{Tree)} \\
= & \pi_2(\text{hd}(\{1, T\})) \\
= & \text{(def. } \pi_2 \text{ and hd)} \\
T
\end{align*}$$

A.8.2 Proof of $(c \downarrow i) \uparrow = c$

Take $c = (s, (a, \tau)) : c'$ where $i \in \text{inds } \tau$:

$$\begin{align*}
((s, (a, \tau)) : c') \downarrow i \\
= & \text{(def. of } \downarrow) \\
(\bar{i}, \tau[i]) : (s, (a, \tau)) : c' \\
= & \text{(def. of } \uparrow) \\
(s, (a, \tau)) : c'
\end{align*}$$

A.8.3 Proof of $(\bar{i}c) \in \{1\ldots\#(c \uparrow)\}$

The precondition for $\uparrow$ requires that $\text{len } c > 1$. So let $c = (s_1, \tau_2[s_1]) : (s_2, (a_2, \tau_2)) : c'$ in

$$\begin{align*}
(\bar{i}c) \in \{1\ldots\#(c \uparrow)\} \\
= & \text{(bar } c = s_1) \\
= & \text{(s_1 } \in \{1\ldots\#(c \uparrow)\}) \\
= & \text{(expand } c) \\
= & \text{(def. } \uparrow) \\
= & \text{(len } s_1) \\
= & \text{(def. } \text{inds } \tau_2) \\
= & \text{(inv-Curs } c \text{ asserts the above)} \\
T \text{H}
\end{align*}$$
A.8.4 Proof of \((c \uparrow) \downarrow (\lceil c \rceil) = c\)

The precondition for \(\uparrow\) requires \(\text{len}(c) > 1\), so let

\[ c = (s_1, \tau_2[s_1]) : (s_2, (\alpha_2, \tau_2)) : c' \]

We note also that

\[ \lceil c \rceil = s_1 \]

We reduce the lhs:

\[
\begin{align*}
(c \uparrow) \downarrow (\lceil c \rceil) \\
= & \text{ (instantiation satisfying pre-\(\uparrow\))} \\
= & \text{ (def. \(\uparrow\))} \\
= & \text{ (def. \(\downarrow\))} \\
= & \text{ (instantiation)} \\
= & c
\end{align*}
\]
A.8.5  **Proof of** $c \Downarrow \downarrow (\#c + 1)$

The precondition for $\uparrow$ requires $\text{len } c > 1$, so let

$$c = (s_1, \tau_2[s_1]) : (s_2, (\alpha_2, \tau_2)) : c'$$

We note also that

$$\#c = s_1$$

The precondition for $\uparrow$ requires that $s_1 + 1 \in \text{inds } \tau_2$.

Reduce the lhs:

$$c \Downarrow \downarrow$$

$\Downarrow$ (instantiation)

$$((s_1, \tau_2[s_1]) : (s_2, (\alpha_2, \tau_2)) : c') \Downarrow$$

$\Downarrow$ (def. $\Downarrow$

$$((s_1 + 1, \tau_2[s_1 + 1]) : (s_2, (\alpha_2, \tau_2)) : c')$$

Reduce the rhs:

$$(c \uparrow) \downarrow (\#c + 1)$$

$\uparrow$ (instantiation satisfying pre-$\uparrow$

$$(((s_1, \tau_2[s_1]) : (s_2, (\alpha_2, \tau_2)) : c') \uparrow \downarrow (s_1 + 1)$$

$\uparrow$ (def. $\uparrow$

$$((s_2, (\alpha_2, \tau_2)) : c') \downarrow (s_1 + 1)$$

$\downarrow$ (def. $\downarrow$

$$((s_1 + 1, \tau_2[s_1 + 1]) : (s_2, (\alpha_2, \tau_2)) : c')$$

The two reductions give identical results. ♦
A.8.6 Proof of \( (c \downarrow) \uparrow = c \uparrow \)

The precondition for \( \uparrow \) requires \( \text{len } c > 1 \), so let

\[
c = (s_1, \tau_2[s_1]) : (s_2, (a_2, \tau_2)) : c'
\]

The precondition for \( \uparrow \) requires that \( s_1 + 1 \in \text{inds } \tau_2 \).

Reduce the lhs:

\[
\begin{align*}
(c \downarrow) \uparrow \\
&= \text{(instantiation)} \\
&= (((s_1, \tau_2[s_1]) : (s_2, (a_2, \tau_2)) : c') \uparrow) \uparrow \\
&= \text{(def. \( \downarrow \))} \\
&= (s_1 + 1, \tau_2[s_1 + 1]) : (s_2, (a_2, \tau_2)) : c' \uparrow \\
&= \text{(def. \( \uparrow \))} \\
&= (a_2, (a_2, \tau_2)) : c'
\end{align*}
\]

Reduce the rhs:

\[
\begin{align*}
c \uparrow \\
&= \text{(instantiation satisfying pre-\( \uparrow \))} \\
&= ((s_1, \tau_2[s_1]) : (s_2, (a_2, \tau_2)) : c') \uparrow \\
&= \text{(def. \( \uparrow \))} \\
&= (a_2, (a_2, \tau_2)) : c'
\end{align*}
\]

The two reductions give identical results. ✤
A.8.7 Proof of $L (\alpha \mathfrak{c}) = \text{True}$

We prove this by strong induction on the height of $\mathfrak{c}$:

$$P(n) \equiv h(\mathfrak{c}) = n \land L (\alpha \mathfrak{c})$$

Base case: $h(\mathfrak{c}) = 1$
Take $h(\mathfrak{c}) = 1$ as an assumption.

$$L (\alpha \mathfrak{c}) = (\text{defn. alpha})$$
$$L (L \mathfrak{c} \rightarrow \mathfrak{c}, \alpha(\mathfrak{c} \downarrow 1)) = (\text{prop. } h(\mathfrak{c}) = 1 \equiv L \mathfrak{c}, \text{conditional})$$
$$L \mathfrak{c} = (\text{prop. } h(\mathfrak{c}) = 1 \equiv L \mathfrak{c})$$
$$\text{TRUE}$$

Inductive Case: $h(\mathfrak{c}) = n$
We assume the property holds for all $\mathfrak{c}'$ where $h(\mathfrak{c}') < n$. We then show it will also hold for $\mathfrak{c}$. So, we assume

$$h(\mathfrak{c}') < n \Rightarrow L (\alpha \mathfrak{c}')$$

$$L (\alpha \mathfrak{c}) = (\text{defn. alpha})$$
$$L (L \mathfrak{c} \rightarrow \mathfrak{c}, \alpha(\mathfrak{c} \downarrow 1))$$

We do a case split on $L \mathfrak{c}$.

Case 1: $L \mathfrak{c}$
The proof proceeds as per the $h(\mathfrak{c}) = 1$ case from here.

Case 2: $\neg L \mathfrak{c}$

$$L (L \mathfrak{c} \rightarrow \mathfrak{c}, \alpha(\mathfrak{c} \downarrow 1)) = (\neg L \mathfrak{c}, \text{conditional})$$
$$L (\alpha(\mathfrak{c} \downarrow 1)) = (\text{prop. } h(\mathfrak{c} \downarrow i) < h(\mathfrak{c}), \text{assumption, modus ponens})$$
$$\text{True}$$

A.8.8 Proof of $L (\mathfrak{c} \uparrow) = \text{True}$

$$L (\mathfrak{c} \uparrow) = (\text{def. } \uparrow)$$
$$L (\alpha((\mathfrak{c} \uparrow) \uparrow)) = (\text{Property } L (\alpha \mathfrak{c}') \text{ with } \mathfrak{c}' = (\mathfrak{c} \uparrow) \uparrow)$$
$$\text{TRUE}$$
A.8.9  **Proof of** $R\ c \Rightarrow \Omega\ c$

Let $c = (s, (a, \tau)) : c'$ in

\[
R((s, (a, \tau)) : c')
\]

= (def. $R$)

\[
((s, (a, \tau)) : c') = \langle 1, \_ \rangle
\]

⇒ (pattern matching)

\[
c = \langle 1, (a, \tau) \rangle
\]

= (func. distr. over $=$)

\[
\Omega\ c = \Omega(\langle 1, (a, \tau) \rangle)
\]

= (defn. of $\Omega$)

\[
\Omega\ c = R(\langle 1, (a, \tau) \rangle) \rightarrow \text{TRUE} , \ldots
\]

= (defn. of $R$)

\[
\Omega\ c = \langle 1, (a, \tau) \rangle = \langle 1, \_ \rangle \rightarrow \text{TRUE} , \ldots
\]

⇒ (pattern matching)

\[
\Omega\ c = \text{TRUE} \rightarrow \text{TRUE} , \ldots
\]

= (conditional)

\[
\Omega\ c = \text{TRUE}
\]

= (prop. calc.)

\[
\Omega\ c
\]

\[\blacklozenge\]

A.8.10  **Proof of** $R(C\text{ars}(\text{Tree}\ c)) = \text{True}$

Let $c = (s, (a, \tau)) : c'$ in

\[
R(\text{C\text{ars}}(\text{Tree}((s, (a, \tau)) : c')))\]

= (def. of $\text{Tree}$)

\[
R(\text{C\text{ars}}((a, \tau)))
\]

= (def. of $\text{C\text{ars}}$)

\[
R(\langle 1, (a, \tau) \rangle)
\]

= (def. of $R$)

\[
\langle 1, (a, \tau) \rangle = \langle 1, \_ \rangle
\]

⇒ (pattern matching)

\[
\text{TRUE}
\]