Department of Information and Computing Sciences
Utrecht University

INFOAFP – Exam

Andres Löh

Monday, 19 April 2010, 09:00–12:00

Solutions

- Not all possible solutions are given.
- In many places, much less detail than I have provided in the example solution was actually required.
- Solutions may contain typos.

Zippers (33 points total)

A *zipper* is a data structure that allows navigation in another tree-like structure. Consider binary trees:

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
  deriving (Eq, Show)
```

A *one-hole context* for trees is given by the following datatype:

```
data TreeCtx a = NodeL () (Tree a) | NodeR (Tree a) ()
  deriving (Eq, Show)
```

The idea is as follows: leaves contain no subtrees, therefore they do not occur in the context type. In a node, we can focus on either the left or the right subtree. The context then consists of the other subtree. The use of () is just to mark the position of the hole – it is not really needed.

We can plug a tree into the hole of a context as follows:

```
plugTree :: Tree a -> TreeCtx a -> Tree a
plugTree l (NodeL () r) = Node l r
plugTree r (NodeR l ()) = Node l r
```

A zipper for trees encodes a tree where a certain subtree is currently in focus. Since the focused tree can be located deep in the full tree, one element of type *TreeCtx a* is not sufficient. Instead, we store the focused subtree together with a *list* of one-layer contexts that encodes the path from the focus to the root node:

```
data TreeZipper a = TZ (Tree a) [TreeCtx a]
  deriving (Eq, Show)
```

We can recover the full tree from the zipper as follows:

```
leave :: TreeZipper a -> Tree a
leave (TZ t cs) = foldl plugTree t cs
```

Consider the tree

```
tree :: Tree Char
tree = Node (Node (Leaf 'a') (Leaf 'b'))
            (Node (Leaf 'c') (Leaf 'd'))
```

If we focus on the rightmost leaf containing `'d'`, the corresponding zipper structure is

```
example :: TreeZipper Char
example = TZ (Leaf 'd')
             [NodeR (Leaf 'c') (), NodeR (Node (Leaf 'a') (Leaf 'b')) ()]
```

1 (3 points). Define a function

$$enter :: Tree\ a \rightarrow TreeZipper\ a$$

that creates a zipper from a tree such that the full tree is in focus. •

Solution 1.

$$enter\ t = TZ\ t\ [\,]$$

◦

Moving the focus from a tree down to the left subtree works as follows:

$$\begin{aligned}
&down :: TreeZipper\ a \rightarrow Maybe\ (TreeZipper\ a)\\
&down\ (TZ\ (Leaf\ x)\quad cs) = Nothing\\
&down\ (TZ\ (Node\ l\ r)\ cs) = Just\ (TZ\ l\ (NodeL\ ()\ r : cs))
\end{aligned}$$

The function fails if there is no left subtree, i. e., if we are in a leaf.

2 (8 points). Define functions

$$\begin{aligned}
&up\quad :: TreeZipper\ a \rightarrow Maybe\ (TreeZipper\ a)\\
&right :: TreeZipper\ a \rightarrow Maybe\ (TreeZipper\ a)
\end{aligned}$$

that move the focus from a subtree to its parent node or to its right sibling, respectively. Both functions should fail (by returning *Nothing*) if the move is not possible. •

Solution 2. These are the simple definitions:

$$\begin{aligned}
&up\quad (TZ\ t\ (c : cs)\qquad\qquad) = Just\ (TZ\ (plugTree\ t\ c)\ cs)\\
&up\quad _\qquad\qquad\qquad\qquad\ \ = Nothing\\
&right\ (TZ\ l\ (NodeL\ ()\ r : cs)) = Just\ (TZ\ r\ (NodeR\ l\ () : cs))\\
&right\ _\qquad\qquad\qquad\qquad\ \ = Nothing
\end{aligned}$$

The function *right* fails if there is no immediate right sibling. If we want to move to the *right* even if there is no immediate sibling, we can define

$$\begin{aligned}
&right' :: TreeZipper\ a \rightarrow Maybe\ (TreeZipper\ a)\\
&right'\ z = right\ z\ \text{`}mplus\text{`}\ (up\ z \gg= right' \gg= down)
\end{aligned}$$

◦

3 (6 points). Assuming a suitable instance

$$\textbf{instance}\ Arbitrary\ a \Rightarrow Arbitrary\ (TreeZipper\ a)$$

consider the QuickCheck property

$$\begin{aligned}
&downUp :: (Eq\ a) \Rightarrow TreeZipper\ a \rightarrow Bool\\
&downUp\ z = (down\ z \gg= up) == Just\ z
\end{aligned}$$

Give a counterexample for this property, and suggest how the property can be improved so that the test will pass. •

4 (4 points)**.** Is

$$\begin{aligned}&\mathit{left} :: \mathit{TreeZipper}\ a \to \mathit{Maybe}\ (\mathit{TreeZipper}\ a)\\&\mathit{left}\ z = \mathit{up}\ z \gg\!= \mathit{down}\end{aligned}$$

a suitable definition for *left*? Give reasons for your answer. [No more than 30 words.] •

Solution 4. It moves to the left sibling when possible. In the root, *left* fails (which is fine). In other nodes without left siblings, *left* returns to the same place. ◦

5 (6 points)**.** The concept of a *one-hole context* is not limited to binary trees. Give a suitable definition of *ListCtx* such that we can define

$$\textbf{data}\ \mathit{ListZipper}\ a = \mathit{LZ}\ [a]\ [\mathit{ListCtx}\ a]$$

and in principle play the same game as with the zipper for trees. Also define the function

$$\mathit{plugList} :: [a] \to \mathit{ListCtx}\ a \to [a]$$

the combines a list context with a list. •

Solution 5. There is no way to descend into an empty list, and only one way to descend into a non-empty list. When descending to the tail, we have to remember the element we pass, so the list context contains a single element:

$$\textbf{type}\ \mathit{ListCtx}\ a = a$$

Plugging is just cons-ing:

$$\mathit{plugList} = \mathit{flip}\ (:)$$

◦

6 (6 points)**.** Discuss the necessity of *up*, *down*, *left* and *right* functions for the *ListZipper*, and describe what they would do. No need to define them (although it is ok to do so). [No more than 40 words.] •

Solution 6. The functions *up* and *down* correspond to moving left and right in the list, respectively:

$$\begin{aligned}&\mathit{up} && :: \mathit{ListZipper}\ a \to \mathit{Maybe}\ (\mathit{ListZipper}\ a)\\&\mathit{up} && (\mathit{LZ}\ \mathit{xs}\ (c : \mathit{cs})) = \mathit{Just}\ (\mathit{LZ}\ (c : \mathit{xs})\ \mathit{cs})\\&\mathit{up} && \_ \qquad\qquad\qquad\quad = \mathit{Nothing}\\&\mathit{down} && :: \mathit{ListZipper}\ a \to \mathit{Maybe}\ (\mathit{ListZipper}\ a)\\&\mathit{down} && (\mathit{LZ}\ (x : \mathit{xs})\ \mathit{cs}) = \mathit{Just}\ (\mathit{LZ}\ \mathit{xs}\ (x : \mathit{cs}))\\&\mathit{down} && \_ \qquad\qquad\qquad\quad = \mathit{Nothing}\end{aligned}$$

The functions *left* and *right* are not needed, as there are no siblings in the case of lists. ◦

Type isomorphisms (12 points total)

7 (6 points)**.** A different definition for one-hole contexts of trees is the following:

$$\textbf{data}\ Dir = L \mid R$$
$$\textbf{type}\ TreeCtx'\ a = (Dir, Tree\ a)$$

Show that, ignoring undefined values, the types *TreeCtx* and *TreeCtx′* are isomorphic, by giving conversion functions and stating the properties that the conversion functions must adhere to (*no proofs required*). •

Solution 7. The conversion functions are:

$$\begin{aligned}
&from :: TreeCtx\ a \to TreeCtx'\ a\\
&from\ (NodeL\ ()\ r) = (L, r)\\
&from\ (NodeR\ l\ ()) = (R, l)\\
&to :: TreeCtx'\ a \to TreeCtx\ a\\
&to\ (L, r) = NodeL\ ()\ r\\
&to\ (R, l) = NodeR\ l\ ()
\end{aligned}$$

The conversion functions must be mutual inverses:

$$\begin{aligned}
&\forall (c :: TreeCtx\ \ a). \quad to\ (from\ c) \equiv c\\
&\forall (c :: TreeCtx'\ a). \quad from\ (to\ c) \equiv c
\end{aligned}$$

It is very easy to see that these properties hold. ◦

8 (6 points)**.** In Haskell's lazy setting, how many different values are there of type *TreeCtx Bool* if we restrict the occurrences of *Tree Bool* to be leaves. And how many different values are there of type *TreeCtx′ Bool* given the same restriction? (Hint: note that the use of () in the definition of *TreeCtx* is relevant here.) •

Solution 8. For *TreeCtx Bool* there are thirteen (or seventeen):

- $\bot$,
- for *NodeL*, there are six:
  $NodeL\ \bot\ (Leaf\ \bot)$, $NodeL\ ()\ (Leaf\ \bot)$, $NodeL\ \bot\ (Leaf\ True)$, $NodeL\ \bot\ (Leaf\ False)$, $NodeL\ ()\ True$, $NodeL\ ()\ False$,
- and analogously, we get six for *NodeR*.

It is also ok to count $NodeL\ \bot\ \bot$, $NodeL\ ()\ \bot$, $NodeR\ \bot\ \bot$ and $NodeR\ ()\ \bot$.

For *TreeCtx′ Bool* there are ten (or thirteen): $\bot$, $(\bot, Leaf\ \bot)$, $(L, Leaf\ \bot)$, $(R, Leaf\ \bot)$, $(\bot, Leaf\ True)$, $(\bot, Leaf\ False)$, $(L, Leaf\ True)$, $(L, Leaf\ False)$, $(R, Leaf\ True)$, $(R, Leaf\ False)$. If you counted the extra values before, then we should count $(\bot, \bot)$, $(L, \bot)$, and $(R, \bot)$ here as well. ◦

Lenses (14 points total, plus 5 bonus points)

A so-called *lens* is (among other things) a way to access a substructure of a larger structure by grouping a function to extract the substructure with a function to update the substructure:

$$\begin{aligned}\textbf{data}\ a \mapsto b = \mathit{Lens}\ \{&\mathit{extract} :: a \to b,\\ &\mathit{insert}\ \ :: b \to a \to a\}\end{aligned}$$

(We assume here that we enable infix type constructors, and that $\mapsto$ is a valid symbol for such a constructor.)

Lenses are supposed to adhere to the following two *extract*/*insert* laws:

$$\begin{array}{ll}\forall (f :: a \mapsto b)\ (x :: a). & \mathit{insert}\ f\ (\mathit{extract}\ f\ x)\ x \equiv x\\ \forall (f :: a \mapsto b)\ (x :: b)\ (y :: a). & \mathit{extract}\ f\ (\mathit{insert}\ f\ x\ y) \equiv x\end{array}$$

A trivial lens is the identity lens that returns the complete structure:

$$\begin{aligned}&\mathit{idLens} :: a \mapsto a\\ &\mathit{idLens} = \mathit{Lens}\ \{\mathit{extract} = \mathit{id}, \mathit{insert} = \mathit{const}\}\end{aligned}$$

It is trivial to see that *idLens* fulfills the two laws.

9 (4 points). Define a lens that accesses the focus component of a tree zipper structure:

$$\mathit{focus} :: \mathit{TreeZipper}\ a \mapsto \mathit{Tree}\ a$$

•

Solution 9.

$$\begin{aligned}\mathit{focus} = \mathit{Lens}\ \{&\mathit{extract} = \lambda (\mathit{TZ}\ t\ \mathit{cs}) \quad\ \to t,\\ &\mathit{insert}\ \ = \lambda t\ (\mathit{TZ}\ \_\ \mathit{cs}) \to \mathit{TZ}\ t\ \mathit{cs}\}\end{aligned}$$

○

10 (4 points). Define a function that updates the substructure accessed by a lens according to the given function:

$$\mathit{update} :: (a \mapsto b) \to (b \to b) \to (a \to a)$$

•

Solution 10.

$$\mathit{update}\ (\mathit{Lens}\ \mathit{ext}\ \mathit{ins})\ f\ x = \mathit{ins}\ (f\ (\mathit{ext}\ x))\ x$$

○

Lenses can be composed. Structures that support identity and composition are captured by the following type class:

$$
\begin{array}{l}
\textbf{class}\ \mathit{Category}\ \mathit{cat}\ \textbf{where} \\
\quad \mathit{id}\ \ \ :: \mathit{cat}\ a\ a \\
\quad (\circ) :: \mathit{cat}\ b\ c \rightarrow \mathit{cat}\ a\ b \rightarrow \mathit{cat}\ a\ c
\end{array}
$$

For instance, functions are an instance of the category class, with the usual definitions of identity and function composition:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Category}\ (\rightarrow)\ \textbf{where} \\
\quad \mathit{id}\ \ \ = \mathit{Prelude.id} \\
\quad (\circ) = (\mathit{Prelude}.\circ)
\end{array}
$$

11 (6 points)**.** Define an instance of the *Category* class for lenses:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Category}\ (\mapsto)\ \textbf{where} \\
\quad \ldots
\end{array}
$$

•

Solution 11.

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Category}\ (\mapsto)\ \textbf{where} \\
\quad \mathit{id} = \mathit{idLens} \\
\quad (\circ)\ f\ g = \\
\qquad\qquad \mathit{Lens}\ \{\mathit{extract} = \mathit{extract}\ f \circ \mathit{extract}\ g, \\
\qquad\qquad\qquad\ \ \mathit{insert}\ \ = \mathit{update}\ g \circ \mathit{insert}\ f\}
\end{array}
$$

∘

12 (5 *bonus points*)**.** Prove using equational reasoning that if the two *extract*/*insert* laws stated above hold for both f and g, then they also hold for $f \circ g$. •

Solution 12. Let $x :: a, f :: b \mapsto c, g :: a \mapsto b$.

$$
\begin{array}{ll}
& \mathit{insert}\ (f \circ g)\ (\mathit{extract}\ (f \circ g)\ x)\ x \\
\equiv & \qquad \{\ \text{definition of } \mathit{insert}\ \} \\
& (\mathit{update}\ g \circ \mathit{insert}\ f)\ (\mathit{extract}\ (f \circ g)\ x)\ x \\
\equiv & \qquad \{\ \text{definition of } (\circ)\ \} \\
& \mathit{update}\ g\ (\mathit{insert}\ f\ (\mathit{extract}\ (f \circ g)\ x))\ x \\
\equiv & \qquad \{\ \text{definition of } \mathit{update}\ \} \\
& \mathit{insert}\ g\ (\mathit{insert}\ f\ (\mathit{extract}\ (f \circ g)\ x)\ (\mathit{extract}\ g\ x))\ x \\
\equiv & \qquad \{\ \text{definition of } \mathit{extract}\ \} \\
& \mathit{insert}\ g\ (\mathit{insert}\ f\ ((\mathit{extract}\ f \circ \mathit{extract}\ g)\ x)\ (\mathit{extract}\ g\ x))\ x
\end{array}
$$

$$
\begin{aligned}
\equiv&\quad \{\text{ definition of } (\circ) \}\\
&\mathit{insert}\ g\ (\mathit{insert}\ f\ (\mathit{extract}\ f\ (\mathit{extract}\ g\ x))\ (\mathit{extract}\ g\ x))\ x\\
\equiv&\quad \{\text{ assumption on } f \}\\
&\mathit{insert}\ g\ (\mathit{extract}\ g\ x)\ x\\
\equiv&\quad \{\text{ assumption on } g \}\\
&x
\end{aligned}
$$

Now let $x :: b$, $y :: a$, $f :: b \mapsto c$ and $g :: a \mapsto b$.

$$
\begin{aligned}
&\mathit{extract}\ (f \circ g)\ (\mathit{insert}\ (f \circ g)\ x\ y)\\
\equiv&\quad \{\text{ definition of } \mathit{extract} \}\\
&(\mathit{extract}\ f \circ \mathit{extract}\ g)\ (\mathit{insert}\ (f \circ g)\ x\ y)\\
\equiv&\quad \{\text{ definition of } (\circ) \}\\
&\mathit{extract}\ f\ (\mathit{extract}\ g\ (\mathit{insert}\ (f \circ g)\ x\ y))\\
\equiv&\quad \{\text{ definition of } \mathit{insert} \}\\
&\mathit{extract}\ f\ (\mathit{extract}\ g\ ((\mathit{update}\ g \circ \mathit{insert}\ f)\ x\ y))\\
\equiv&\quad \{\text{ definition of } (\circ) \}\\
&\mathit{extract}\ f\ (\mathit{extract}\ g\ (\mathit{update}\ g\ (\mathit{insert}\ f\ x)\ y))\\
\equiv&\quad \{\text{ definition of } \mathit{update} \}\\
&\mathit{extract}\ f\ (\mathit{extract}\ g\ (\mathit{insert}\ g\ (\mathit{insert}\ f\ x\ y)\ y)\\
\equiv&\quad \{\text{ assumption on } g \}\\
&\mathit{extract}\ f\ (\mathit{insert}\ f\ x\ y)\\
\equiv&\quad \{\text{ assumption on } f \}\\
&x
\end{aligned}
$$

∘

Monad transformers (22 points total)

Consider the monad *TraverseTree*, defined as follows:

type *TraverseTree a* = *StateT* (*TreeZipper a*) *Maybe*

13 (3 points)**.** What is the kind of *TraverseTree*? •

Solution 13.

$* \to * \to *$

∘

14 (6 points). Define a function

$$nav :: (TreeZipper\ a \rightarrow Maybe\ (TreeZipper\ a)) \rightarrow TraverseTree\ a\ ()$$

that turns a navigation function like *down*, *up*, or *right* into a monadic operation on *TraverseTree*. •

Solution 14.

```
nav f =
  do
    l ← get
    x ← lift $ f l
    put x
```

Note that using *modify* is problematic, because we cannot lift the argument to *modify* into the outer monad. ◦

Given a lense and the *MonadState* interface, we can define useful helpers to access parts of the monadic state:

```
getLens :: MonadState s m ⇒ (s ↦ a) → m a
getLens f = gets (extract f)
putLens :: MonadState s m ⇒ (s ↦ a) → a → m ()
putLens f x = modify (insert f x)
modifyLens :: MonadState s m ⇒ (s ↦ a) → (a → a) → m ()
modifyLens f g = modify (update f g)
```

We can now define the following piece of code:

```
ops :: TraverseTree Char ()
ops =
  do
    nav down
    x ← getLens focus
    nav right
    putLens focus x
    nav down
    modifyLens focus (const $ Leaf 'X')
```

15 (6 points). Given all the functions so far and once again tree

```
tree = Node (Node (Leaf 'a') (Leaf 'b'))
            (Node (Leaf 'c') (Leaf 'd'))
```

what is the result of evaluating the following declaration:

```
test = leave (snd (fromJust (runStateT ops (enter tree))))
```

•

Solution 15. The result is

$$Node\ (Node\ (Leaf\ \texttt{'a'})\ (Leaf\ \texttt{'b'}))\ (Node\ (Leaf\ \texttt{'X'})\ (Leaf\ \texttt{'b'}))$$

○

16 (7 points)**.** Explain how a compiler based on passing dictionaries for type classes can construct the dictionary to pass to the *modifyLens* call in the last line of the definition of *ops* above. •

Solution 16. The call to modifyLens requires an instance

$$MonadState\ (TreeZipper\ Char)\ (TraverseTree\ Char)$$

which after expanding the type synonym means

$$MonadState\ (TreeZipper\ Char)\ (StateT\ (TreeZipper\ Char)\ Maybe)$$

Reading classes as dictionary types, we thus need a dictionary of the type above. We have the instances

instance *Monad Maybe*
instance *Monad m* $\Rightarrow$ *MonadState s* (*StateT s m*)

available, in other words, we can assume dictionaries:

monadMaybe :: *Monad Maybe*
monadState :: *Monad m* $\rightarrow$ *MonadState s* (*StateT s m*)

The desired dictionary can thus be constructed by using

monadState monadMaybe

○

Trees, shapes and pointers in Agda (19 points total)

Consider the definitions of *List*, $\mathbf{N}$, *Vec* and *Fin* in Agda. These four types are related as follows:

Natural numbers describe the *shapes* of lists (if we instantiate the element type of lists to the unit type, we obtain a type isomorphic to the natural numbers). Indexing lists by their shapes yields vectors. Finally, *Fin* is the type of *pointers* into vectors such that we can define a safe lookup function.

Now consider binary trees (as before), given in Agda by:

data *Tree* (*A* : *Set*) : *Set* **where**
  leaf : *A* $\rightarrow$ *Tree A*
  node : *Tree A* $\rightarrow$ *Tree A* $\rightarrow$ *Tree A*

The type of shapes for trees is given by:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Shape} : \mathit{Set}\ \textbf{where} \\
\quad \mathit{end}\ \ : \mathit{Shape} \\
\quad \mathit{split} : \mathit{Shape} \to \mathit{Shape} \to \mathit{Shape}
\end{array}
$$

17 (5 points)**.** Define a datatype *STree* of shape-indexed binary trees (i. e., *STree* corresponds to *Vec*):

$$
\begin{array}{l}
\textbf{data}\ \mathit{STree}\ (A : \mathit{Set}) : \mathit{Shape} \to \mathit{Set}\ \textbf{where} \\
\quad \ldots
\end{array}
$$

•

Solution 17.

$$
\begin{array}{l}
\textbf{data}\ \mathit{STree}\ (A : \mathit{Set}) : \mathit{Shape} \to \mathit{Set}\ \textbf{where} \\
\quad \mathit{leaf}\ \ : A \to \mathit{STree}\ A\ \mathit{end} \\
\quad \mathit{node} : \forall \{s\ t\} \to \mathit{STree}\ A\ s \to \mathit{STree}\ A\ t \to \mathit{STree}\ A\ (\mathit{split}\ s\ t)
\end{array}
$$

○

18 (6 points)**.** Define a datatype *Path* of shape-indexed pointers (i. e., *Path* corresponds to *Fin*):

$$
\begin{array}{l}
\textbf{data}\ \mathit{Path} : \mathit{Shape} \to \mathit{Set}\ \textbf{where} \\
\quad \ldots
\end{array}
$$

Note that a value p of type *Path* s should point to an element in a tree of shape s. •

Solution 18.

$$
\begin{array}{l}
\textbf{data}\ \mathit{Path} : \mathit{Shape} \to \mathit{Set}\ \textbf{where} \\
\quad \mathit{here}\ \ : \mathit{Path}\ \mathit{end} \\
\quad \mathit{left}\ \ \ : \forall \{s\ t\} \to \mathit{Path}\ s \to \mathit{Path}\ (\mathit{split}\ s\ t) \\
\quad \mathit{right} : \forall \{s\ t\} \to \mathit{Path}\ t \to \mathit{Path}\ (\mathit{split}\ s\ t)
\end{array}
$$

○

19 (4 points)**.** Define a function *zipWith* on shape-indexed trees that merges two trees of the same shape and combines the elements according to the given function.

$$
\begin{array}{l}
\mathit{zipWith} : \forall \{A\ B\ C\ s\} \to (A \to B \to C) \to \\
\qquad\qquad \mathit{STree}\ A\ s \to \mathit{STree}\ B\ s \to \mathit{STree}\ C\ s
\end{array}
$$

•

Solution 19.

$$
\begin{aligned}
&\mathit{zipWith}\ f\ (\mathit{leaf}\ x) &&(\mathit{leaf}\ y) &&= \mathit{leaf}\ (f\ x\ y)\\
&\mathit{zipWith}\ f\ (\mathit{node}\ l_1\ r_1) &&(\mathit{node}\ l_2\ r_2) &&= \mathit{node}\ (\mathit{zipWith}\ f\ l_1\ l_2)\ (\mathit{zipWith}\ f\ r_1\ r_2)
\end{aligned}
$$

○

20 (4 points)**.** Define a function *lookup* on shape-indexed trees

$$\mathit{lookup} : \forall \{A\ s\} \to \mathit{STree}\ A\ s \to \mathit{Path}\ s \to A$$

that returns the element stored at the given path. •

Solution 20.

$$
\begin{aligned}
&\mathit{lookup}\ (\mathit{leaf}\ x) &&\mathit{here} &&= x\\
&\mathit{lookup}\ (\mathit{node}\ l\ r) &&(\mathit{left}\ \ p) &&= \mathit{lookup}\ l\ p\\
&\mathit{lookup}\ (\mathit{node}\ l\ r) &&(\mathit{right}\ p) &&= \mathit{lookup}\ r\ p
\end{aligned}
$$

○