Formal Verification of Huet's Zipper in Coq

Nicholas Garcia
Greg Heartsfield

May 6, 2008

Abstract

We present a formally verified implementation of the functional data structure known as Huet's Zipper. The process of translating a standard implementation in Haskell to Coq's specification language Gallina is described, along with the guarantees we were able to provide. Finally, an automatic extraction of the verified program is produced in a separate language, O'Caml.

1 Introduction

Programmers today have a wealth of tools available to assist them in ensuring code meets specifications. Unit testing frameworks provide confidence and verification for specific inputs, and automatic specification-based tools like QuickCheck [2] allow the tests themselves to be generated based on program input types. Model finders such as Alloy exploit the "small scope hypothesis" [1] and demonstrate correctness within certain bounds. At what point can we say our program is correct though? Each of these tools allow us to be more confident than we were before using them, but inevitably the testing must end, and we must provide a caveat for our result. The assurances we then give may be as weak as manually written unit tests for obvious corner-cases, or as strong as an absolute guarantee for all input below a certain size, but there remains input for which we simply cannot make statements about how our program will react towards.

Through strict formal methods however, we do have the power to make some of the broadest statements about correctness. Verified specifications will hold true for any inputs, with the same confidence that we typically reserve for mathematical facts. When users of testing tools decide what they wish to verify, they are very often thinking in terms of general properties, not individual tests, and it is the tool that forces them into concretizing those properties into individual tests. In this sense, formal methods work on a similar level to a testers goals.

One of the strongest and most elusive properties of a program that we desire to know is: *will it terminate*? While that question is answered regularly through

informal human inspection, it would be valuable to have this property as a machine-checked assertion. The halting problem is avoided by the fact that we do not need termination checking for arbitrary programs, and we acknowledge that we cannot make this assertion for every program. Nevertheless, for those programs for which we can make that assertion, it is a very powerful assurance.

The objective of our project is to produce a certified implementation of the zipper, by implementing it in the formal specification language Gallina, proving several properties, and automatically extracting a program that is guaranteed to maintain the proven properties in a more accessible language, O'Caml.

Without a doubt, the advanced testing tools available today are often the pragmatic choice for gaining confidence in program correctness. The unmatched power and generality of the results from formal methods have their place as well, whenever we need to know for sure, instead of simply being highly confident. One aim of this paper, beyond the goal expressed in the title, is to answer questions about how far the divide between these two techniques really is, and how far we can go in utilizing formal techniques to provide strong assurances for practical assertions.

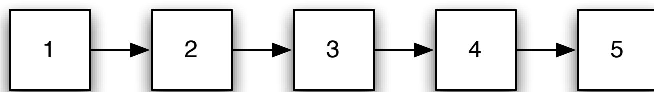
2 The Zipper

Performance is an often-heard concern in pure functional languages, those that require all data structures be immutable. This can often be alleviated by designing structures with immutability in mind, sharing a maximal amount of data after an update, instead of naively copying a structure.

The zipper is a pattern for building data structures that expose some area of focus to modification, while still sharing a large unmodified context [3]. It fixes the problem of cascading modifications present in many standard data structures, such as a linked list, or binary tree.

Looking at a singly linked list, such as the one shown in Figure 1, we can easily illustrate the problem with modification. Say we want to modify the node that currently has a value of 3. To do so without overwriting existing data, we need to create a new node to replace 3, and have it point to the already existing node 4. The preceding node, 2, points to the old 3, so it must be duplicated, and its preceding node must be duplicated, and that is the cause for our poor performance. Modification in this case is $O(n)$, where n is the number of nodes preceding the one to modify.

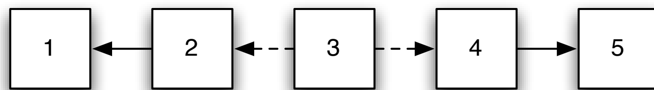
Figure 1: Singly linked list



Contrast this to the situation where there is a node that is "in focus", and

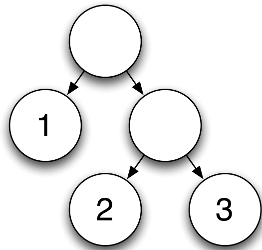
the nodes preceding and after that focus are directed away, as shown in Figure 2. Now, to modify 3, we simply create a new node with a different value, and point it to 2 before, and 4 after. While this is not as simple as directly modifying the value of 3, as one would in a destructive environment, it is an $O(1)$ operation which is significantly better than what we saw previously with the linked list. Operations to move the focus left or right are also $O(1)$, since we can point into any part of the list without need for cascading modifications.

Figure 2: List, with zipper



This concept can be extended to many structures, but it is most often applied to a tree. In this case we mean binary trees with anonymous inner nodes and valuated leaves, as shown in Figure 3. The problem here, as with the linked list, is that modification of a leaf node requires copying the parent node, the parent of that node, and so on.

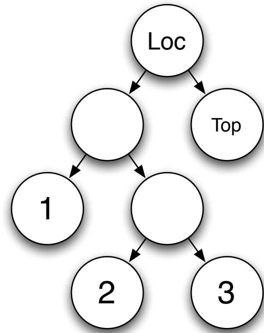
Figure 3: Binary tree



In this case, the zipper structure requires more explanation than that for linked lists, but the concept remains the same. We now have a location node, which is the top-level pointer into the data structure, analagous to the root node of a tree. The location points to two subtrees, one is the current focus of the zipper and the other is the context, which is everything that surrounded that focus in the original tree. The focus is straightforward, being just all or a portion of the original tree. The context, however, is what makes the zipper work. It is essentially an inside-out representation of the rest of the tree, complete with instructions on how to rebuild it from the bottom up. The simplest case is a zipper where the complete original tree is the area of focus, shown in Figure 4.

The context needed to build the original tree is simply a focus that is the top node, which means the context subtree is simply made up of a node labelled

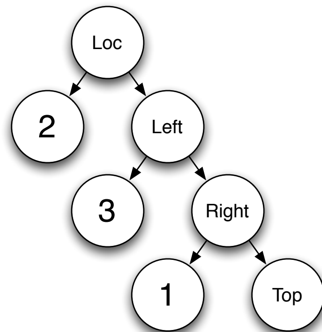
Figure 4: Tree, with zipper



`Top`. We can see at this point that the operation of creating a zipper from a tree can be done in constant time.

A more complete example, where the focus has been shifted from the original tree, is given in Figure 5. This shows a path down the tree that starts with the top root node, moves to the right subtree, and then ends on the left leaf node. In the example we can see this reflected in the context subtree, which can be read as a list of instructions for rebuilding the original tree around the focused leaf node with a value of 2. The labels `Right` and `Left` are the instructions, their left child subtrees are the data, and the final node, `Top`, indicates that the original tree has been restored.

Figure 5: Zipper after moving right and left



Navigation around the zipper is performed by creating new location nodes and a single `Right` or `Left` node to record the path if moving down, or pointing deeper into a context structure if moving up. Thus, navigation around a zipper can occur in constant time, even though it is a comparatively large constant requiring object creation for each movement.

The operations on the zipper that are typically provided and that we are interested in are listed, along with their running time:

- `moveLeft` – move the focus to the left subtree. An $O(1)$ operation.
- `moveRight` – move the focus to the right subtree. An $O(1)$ operation.
- `moveUp` – move the focus to the parent node. An $O(1)$ operation.
- `moveToTop` – move the focus to the top of the tree. $O(n)$, where n is the distance to the top of the tree from the current focus.
- `modify` – apply a given function to the current subtree, and replace it with the result. $O(m)$, where m is the running time of the function provided.
- `locationFromTree` – convert a tree into a zipper. An $O(1)$ operation.
- `treeFromLocation` – convert a zipper into a tree. An $O(1)$ operation.

3 Proof of Termination

One of the notable restrictions of the Gallina language is that all functions specified must be total, and all programs must terminate. This means we achieve a very strong guarantee simply by implementing the zipper in Coq. Unfortunately, this does not come for free, as our implementation is not a straightforward translation from a language like Haskell.

For example, examine the `moveToTop` function in Haskell, shown in Figure 6. Although it is correct as shown, it would not be hard to make a small modification and have a non-terminating program. In fact, merely switching the order of the bottom two lines is sufficient to have a program which can compile but does not terminate, hardly a difficult or uncommon mistake to make.

Figure 6: moveToTop function in Haskell

```
moveToTop :: Location a -> Location a
moveToTop t@(_, Top) = t
moveToTop l = moveToTop (moveUp l)
```

Coq/Gallina solves these types of issues by restricting the form that recursion can take. All recursive arguments must be structurally smaller than the original input to the function. It is easy to see why this is sufficient for termination, as the argument must eventually reach some lower bound and match a base case. In instances where we are dealing with numbers (Coq uses Peano arithmetic, making structure obvious), each recursive call must strip away at least one instance of the successor function.

This presents its own difficulties though, because it is not at all obvious that the result of `(moveUp l)` is structurally smaller than `l` itself. Examining `moveUp`, we see that it doesn't get smaller in any traditional sense, it simply

shuffles arguments around. In this case, we were forced to rely on a technique known as bounded recursion, to impose a numerical measure on the amount of work remaining to do. This measure is computed, and used as a complexity argument which decreases with every call for structural recursion, making it clear to Gallina that the function has no choice but to terminate.

The metric we use for determining the amount of work remaining to perform is the distance from our current focus to the top of the tree. This is more straightforward to define recursively in Coq because of the fact that we will be throwing away data with every call, instead of shuffling it around. Figure 7 demonstrates the pair of functions defined for computing this measure. `distanceToTop_aux` is recursively defined, and returns the distance to reach the `Top` context, while going through structurally smaller instances of the context with each call. `distanceToTop` is a simpler function which simply seeds `distanceToTop_aux` with the starting context and a current count of zero.

Figure 7: distanceToTop function in Gallina/Coq

```
Fixpoint distanceToTop_aux (c:context) (x:nat) {struct c} : nat :=
  match c with
    | top => x
    | leftC d _ => distanceToTop_aux d (x + 1)
    | rightC e _ => distanceToTop_aux e (x + 1)
  end.

Definition distanceToTop (l:location) : nat :=
  match l with
    | mkLoc _ c => distanceToTop_aux c 0
  end.
```

Now that we have a definite bound on the number of recursive calls `moveToTop` will need to do its work, we can write it as a structurally recursive function, with a termination argument of the complexity, as shown in Figure 8. It also is written as a pair of functions, one to hide the initial complexity calculation, the other to implement the structurally recursive logic.

Figure 8: moveToTop function in Gallina/Coq

```
Fixpoint moveToTop_aux (x:nat) (l:location) {struct x} : location :=
  match x with
    | O => l
    | S n => moveToTop_aux n (moveUp l)
  end.

Definition moveToTop (l:location) := moveToTop_aux (distanceToTop l) l.
```

Note that from a complexity perspective, this is an inefficient function, as it

must traverse the entirety of the context twice. Once for the complexity count, and yet another for reconstructing the focus from the entire context.

For the remaining functions in the implementation, only straightforward implementation was required due to the lack of recursion. After this point, we now have one of our strongest guarantees, that all provided functions for the zipper will return a value for any possible input. There is no possibility of infinite loops or runtime errors (with the notable exception of memory exhaustion) in the program.

4 Equivalence of Translation to and from Binary Tree

One very important property we desired to show was that trees could be converted to and from a zipper and remain identical. The first task was to state this as a proposition that Gallina could understand. In Gallina, propositions are represented as type `Prop`, and to prove a proposition, one constructs a valid term that inhabits the type of the proposition. We have existing functions defined for the `treeToLocation` and `locationToTree` functions, but what we want to show is that the composition of these functions behaves as an identity function for trees. That is to say:

$$\forall (t :: Tree),\ t\ =\ (locationToTree \circ treeToLocation)\ t$$

While the truth is fairly obvious upon inspection of the functions, this is a relatively standard property to verify when dealing with data structures that represent trees in alternative ways.

The proof is straightforward and is shown in Figure 9. First, the universal quantifier is replaced by a variable using the `intro` tactic. Then we replace the two functions with their definitions using the `unfold` tactic. These definitions are trivially convertible into an equality with the `change` tactic. Finally, we identify an exact solution using reflexive equality of the `t` term, which completes the proof. We now know that for any possible tree, converting into a zipper and then back into a tree, will result in the same tree.

Figure 9: Proof that trees convert to and from zippers without modification

```
Lemma loc_ident : forall t:tree, eq t (locationToTree (treeToLocation t)).
  intro t.
  unfold locationToTree.
  unfold treeToLocation.
  change (t = t).
  exact (refl_equal t).
Qed.
```

Proofs of this type, showing equality between inductive terms, can sometimes be solved automatically by Coq, using the `auto` tactic, which repeatedly tries

some of the tactics we used in our proof. In fact, for this particular proof, the `auto` tactic really is all that is necessary.

We also compared the specification we just proved with the equivalent QuickCheck property shown in Figure 10. When executed, arbitrary trees are generated and fed into the function as `t`, until a bound is reached or `false` is returned. When compared with the work necessary to prove for all trees, Coq is actually more efficient since QuickCheck required we define a function to generate these arbitrary trees, shown in Figure 11, which in this case is a standard idiom described in the QuickCheck paper [2].

Figure 10: QuickCheck property stating tree equivalence

```
prop_Equiv t = locationToTree (treeToLocation t) == t
```

Figure 11: Arbitrary tree generator in Haskell QuickCheck

```
instance Arbitrary a => Arbitrary (Tree a) where
    arbitrary = sized arbTree
arbTree 0 = liftM Leaf arbitrary
arbTree n = frequency
            [(1, liftM Leaf arbitrary),
             (4, liftM2 Node (arbTree (n `div` 2))
                   (arbTree (n `div` 2)))]
```

5 Proof of Modification Function

Zippers are used for their ability to modify areas of focus in a tree quickly, not just their navigation capabilities, so clearly we are concerned about the correctness of the `modify` function. In this section we present two proofs that confirm our intuition about how `modify` should work.

Our first statement is that using a tree identity function with `modify` returns the original location. This is true since `modify` takes a function and applies it to the area of focus, a tree, and returns a new location. If the function provided returns the same tree, the location as a whole should be identical as well.

This proof, although seemingly straightforward, cannot be solved automatically by Coq. The proof is shown in Figure 12, and is interesting because we can take advantage of the `induction` tactic for solving. The first step is to use `intros` to remove the universal quantification. Now, we have a relatively opaque statement asserting the equality of some location with functions applied to the same location. Location is an inductive type we have defined, and in doing so, we have an induction tactic defined that we can use. Since there is only a single inductive type, `mkLoc`, this reduction leaves us with a single subgoal. Opening the definitions of the `modify` and `tree_ident` functions with `unfold` reduces

both sides of the equality to obviously equivalent statements. The proof is then concluded by using the `apply` tactic with reflexive equality.

Induction is an especially powerful tactic for working with proofs, as it takes advantage of boilerplate functions that are created behind the scenes when we define inductive types. In this case, a function named `location_ind` exists that is used by `induction` to introduce each of the inductive type's identifiers for location, `mkLoc`. Used on the `tree` type, it would produce alternate versions of the current goal for both `Leaf` and `Node`, allowing simplistic proofs on the base case (`Leaf`) to be proven and then extended to all trees.

Figure 12: Proof that applying the identity function with modify preserves location

```
Lemma modify_ident : forall (l:location), eq l (modify l tree_ident).
  intros.
  induction l.
  unfold modify.
  unfold tree_ident.
  apply refl_equal.
Qed.
```

With the next proof, we again introduce something unique and powerful. Instead of using the identity function, we would like to show that when we use *any* function with `modify`, the resulting tree of focus is the same as if we had extracted the focus first and simply applied the same function. This will show that there is no possible way for `modify` to affect anything outside of the area of focus, such as the location itself or the context. To write a property involving an arbitrary function is quite simple; we use universal quantification over the type of valid functions which we can express, like: $\forall\ f :: tree \rightarrow tree$. We can then treat f just as we treated the identity function previously. Using it will make proofs more difficult though, as we cannot simply unfold its definition to get rid of it. In this proof, we will have to clearly show that it exists on both sides of the equality when applied to the same tree argument.

Without going through the proof line by line, that is precisely what was done, as shown in Figure 13. Quantifications were eliminated and the inductive definition of location is expanded using `destruct`. After expanding `getFocus`, it is clear that both sides of the equality are equivalent and the proof can be completed.

While QuickCheck is powerful enough to generate arbitrary functions based on type signatures, we feel the result here is significantly more compelling because of the limits of how many functions can be generated and tested in reasonable time limits and their complexity.

Figure 13: Proof that applying a function with modify is the same as applying it to the focus

```
Lemma modify_subtree_arb : forall (l:location) (f: tree->tree),
      eq (f (getFocus l)) (getFocus (modify l f)).
  unfold tree_ident.
  intros l f.
  destruct l.
  unfold getFocus.
  change (f loc_tree0 = f loc_tree0).
  apply refl_equal.
Qed.
```

6 Proof of Navigation Functions

While the navigation primitives are not complicated functions, we again want to verify that they do what we believe they should. The property we want to show for `moveLeft` and `moveRight` is that after applying the navigation function to a location, the new focus is the same as the subtree of the original focus (right or left, depending on which movement primitive was used). This statement in Gallina and the proof for `moveLeft` (`moveRight` is similar and so is omitted) are shown in Figure 14.

Figure 14: Proof that moveLeft gets left subtree

```
Lemma leftsub : forall (t:tree), eq (getLeftSubtree t)
      (getFocus (moveLeft (treeToLocation t))).
  intro t.
  simpl.
  unfold getFocus.
  unfold getLeftSubtree in |- *.
  destruct t ; apply refl_equal.
Qed.
```

The function `getLeftSubtree` is defined purely on trees and captures the intent of what we would like to have happen on the focused subtree. By unfolding its definition, we alter the left side of the equality to contain just the left subtree. On the right side, unfolding `getFocus` selects the focused subtree after `moveLeft` has modified it to select the left subtree as well. The definition for a tree, being a `leaf` or a `node`, is broken into two separate goals with the `destruct` tactic, which we then finish by applying reflexive equality to each.

We would like to verify one of the less well defined aspects of the `moveUp` function, that when the location points to the top of a tree, calling `moveUp` results in the same location being returned. This doesn't require an in depth transformation, since we can take advantage of the fact that `treeToLocation` gives us a location at the the top, and we are just verifying that `moveUp` has no

effect on this type of input. Indeed, once we write the specification, the `auto` tactic is sufficient to provide the proof.

Figure 15: Proof that moveUp at top returns the same location

```
Lemma trees_convert_to_top : forall (t:tree),
        eq (treeToLocation t) (moveUp (treeToLocation t)).
  auto.
Qed.
```

7 Implementation in Haskell

As a tool in understanding the typical implementation, we have created a zipper in Haskell. Although there is nothing original nor novel about it, we feel it is a necessary benchmark in determining the effort required to produce a standard program compared to one in a specification language like Gallina.

The bulk of the program is similar to the Gallina implementation, except for syntactical differences. For example, compare the `moveLeft` function, shown in Figure 16 for Haskell and in Figure 17 for Gallina.

Figure 16: moveLeft function in Haskell

```
moveLeft :: Location a -> Location a
moveLeft (Node l r, c) = (l, LeftC c r)
moveLeft x = x
```

Figure 17: moveLeft function in Gallina/Coq

```
Definition moveLeft (l:location) :=
  match l with
    mkLoc t c => match t with
                   | leaf a => l
                   | node e f => mkLoc e (leftC c f)
                 end
  end.
```

The required pattern matching is more verbose in Gallina, but the logic is identical. The current node is deconstructed into right and left subtrees, the left subtree becomes the new point of focus, and the right subtree is moved into the surrounding context.

A significant difference mentioned previously between the two languages is that Gallina is total, where in Haskell partial functions can be defined. One implication of this is that our original Haskell program had the possibility of run time errors if given certain inputs. For instance, attempting to use the

`moveLeft` function on a location that was focused on a leaf would cause an error. It was not until we attempted to rewrite the program in Gallina that this was evident. Since the location focus could be either a leaf or a node, Gallina required we handle both cases before it would accept our program as well-typed. Haskell (using the Glasgow Haskell Compiler) accepted the partially defined function and would throw a runtime exception given the wrong input. It was only after we understood Gallina's refusal to accept our code that we could go back and correct the original Haskell program.

Finally, the largest difference experienced was the requirement in Gallina that recursion steps be structurally smaller with each call. The haskell implementation of `moveToTop` is three lines long, and naturally defined. The `moveUp` function is repeatedly called on a location, until the context consists only of `Top`. The equivalent program in Gallina required the addition of a size argument that was used to bound the recursion, as well as a function to calculate the complexity of the operation, `distanceToTop`. The final result was 16 lines of code, not overly burdensome by any means, but it was the largest obstacle to the translation and the proper approach to implementing it was not immediately obvious to us.

8 Program Extraction

While Gallina is a fully capable programming language despite some restrictions on recursion and totality, it is important to be able to have our programs run in more standard environments. We could manually rewrite them, knowing that the algorithms were proven correct, but this leaves the possibility of translation errors. Clearly, it would be valuable to automatically extract programs in different languages, so that we could take advantage both of high performance compilers, and integrate with other software (especially important when we are producing verified data structures).

Coq has facilities for extracting programs to several different languages, including Haskell, O'Caml, and Scheme. Extractions to all three languages were attempted, but only the O'Caml extraction was powerful enough to translate entire Coq modules. Using the other languages would have required us to extract functions out individually, which somewhat negates the advantages of the technique, since no human intervention should be necessary to get the result working.

The entire program extraction is available in Appendix A. Our zipper as developed and verified was parametrically polymorphic with respect to the value held by leaf nodes. Unfortunately, the Coq extraction mechanism could not preserve this, and produced a "hole" in the final code where the concrete type of `coq_A` had to be defined. This is given as `int` in the appendix, but would be up to the user to change as needed (it can be given from within Coq, so direct modification of the generated source is not required). It is a simple modification to restore polymorphism, but as it requires modifying several lines of the generated program, we resisted adding this feature and possibly introducing

errors.

Functionally, the extracted code works as advertised. Identifier naming appears clumsy by default, with the `Coq_` prefix widespread, but this may be preferred in some cases to indicate origin, and can be configured as part of the extraction. More serious is the observation that this is a very direct translation from our Coq/Gallina implementation. The inefficiences we noted in creating recursive functions with proofs of termination are still present. Although it doesn't change the asymptotic running time of any of the functions, it is not ideal.

9 Conclusion

Formal methods are frequently dismissed as being impractical as a tool for the working computer scientist. Our view after this experience is mixed. The goal of producing an automatically extracted program from a formally verified one was achieved, but with significant limitations. The constructivist logic that was necessary to achieve our strong guarantees was not stripped away or optimized in the final result, leading to a significant performance difference from natively written code. This is certainly a rich area for further work, as we see many areas for possible improvement. Notably, the use of Peano arithmetic could be eliminated, resulting in significant space and performance improvements, without violating correctness.

Most promising was the fact that bugs were found, and guarantees could be made, even before specifications were validated. Programming in a total, strongly normalizing language like Gallina was not overly challenging, and forced us to write code that terminated, and was free of run-time exceptions. For providing assurances to programs, this may be the sweet spot balancing effort and results.

As for proving specifications, as opposed to testing them, this was a mixed result as well. Working with proof tactics was rewarding and even fun, more like solving a logic puzzle than programming. We failed to reveal any bugs in code that had gone through specification-based testing, leading us to acknowledge the small-scope hypothesis is valid.

Creating programs that provably adhere to specifications is by no means easy, however it is not insurmountable for inexperienced users to produce strong results. Additionally, some of the most compelling results are available without expending significant effort. We have shown, using Huet's Zipper, that the task of creating real verified software is possible and mature, although it is clear that improvements could be made. Our final recommendation is that formal and informal techniques are complementary; formal methods are not necessarily overly burdensome, and informal techniques can provide very strong assurances. Programmers should not limit themselves to just one type of technique in this important area, but should recognize the unique strengths and weaknesses of each.

References

[1] A. Andoni, D. Daniliuc, S. Khurshid, and D. Marinov. Evaluating the small scope hypothesis.

[2] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279, New York, NY, USA, 2000. ACM.

[3] Gérard Huet. The zipper. *J. Funct. Program.*, 7(5):549–554, 1997.

A O'Caml Extraction

```
type nat =
  | O
  | S of nat

(** val plus : nat -> nat -> nat **)

let rec plus n m =
  match n with
    | O -> m
    | S p -> S (plus p m)

module BinaryTreeZipper =
 struct
  type coq_A = int


  type tree =
    | Coq_leaf of coq_A
    | Coq_node of tree * tree

  (** val tree_rect : (coq_A -> 'a1) -> (tree -> 'a1 -> tree -> 'a1 -> 'a1)
                      -> tree -> 'a1 **)

  let rec tree_rect f f0 = function
    | Coq_leaf a -> f a
    | Coq_node (t0, t1) -> f0 t0 (tree_rect f f0 t0) t1 (tree_rect f f0 t1)

  (** val tree_rec : (coq_A -> 'a1) -> (tree -> 'a1 -> tree -> 'a1 -> 'a1) ->
                     tree -> 'a1 **)

  let rec tree_rec f f0 = function
    | Coq_leaf a -> f a
```

```
  | Coq_node (t0, t1) -> f0 t0 (tree_rec f f0 t0) t1 (tree_rec f f0 t1)

type context =
  | Coq_top
  | Coq_leftC of context * tree
  | Coq_rightC of context * tree

(** val context_rect : 'a1 -> (context -> 'a1 -> tree -> 'a1) -> (context
                       -> 'a1 -> tree -> 'a1) -> context -> 'a1 **)

let rec context_rect f f0 f1 = function
  | Coq_top -> f
  | Coq_leftC (c0, t) -> f0 c0 (context_rect f f0 f1 c0) t
  | Coq_rightC (c0, t) -> f1 c0 (context_rect f f0 f1 c0) t

(** val context_rec : 'a1 -> (context -> 'a1 -> tree -> 'a1) -> (context ->
                      'a1 -> tree -> 'a1) -> context -> 'a1 **)

let rec context_rec f f0 f1 = function
  | Coq_top -> f
  | Coq_leftC (c0, t) -> f0 c0 (context_rec f f0 f1 c0) t
  | Coq_rightC (c0, t) -> f1 c0 (context_rec f f0 f1 c0) t

type location = { loc_tree : tree; loc_ctx : context }

(** val location_rect : (tree -> context -> 'a1) -> location -> 'a1 **)

let location_rect f l =
  let { loc_tree = x; loc_ctx = x0 } = l in f x x0

(** val location_rec : (tree -> context -> 'a1) -> location -> 'a1 **)

let location_rec f l =
  let { loc_tree = x; loc_ctx = x0 } = l in f x x0

(** val loc_tree : location -> tree **)

let loc_tree x = x.loc_tree

(** val loc_ctx : location -> context **)

let loc_ctx x = x.loc_ctx

(** val treeToLocation : tree -> location **)

let treeToLocation t =
```

```
  { loc_tree = t; loc_ctx = Coq_top }

(** val moveLeft : location -> location **)

let moveLeft l =
  let { loc_tree = t; loc_ctx = c } = l in
  (match t with
     | Coq_leaf a -> l
     | Coq_node (e, f) -> { loc_tree = e; loc_ctx = (Coq_leftC (c, f)) })

(** val moveRight : location -> location **)

let moveRight l =
  let { loc_tree = t; loc_ctx = c } = l in
  (match t with
     | Coq_leaf a -> l
     | Coq_node (e, f) -> { loc_tree = f; loc_ctx = (Coq_rightC (c, e)) })

(** val moveUp : location -> location **)

let moveUp l =
  let { loc_tree = t; loc_ctx = c } = l in
  (match c with
     | Coq_top -> l
     | Coq_leftC (e, f) -> { loc_tree = (Coq_node (t, f)); loc_ctx = e }
     | Coq_rightC (g, h) -> { loc_tree = (Coq_node (h, t)); loc_ctx = g })

(** val getFocus : location -> tree **)

let getFocus l =
  l.loc_tree

(** val modify : location -> (tree -> tree) -> location **)

let modify l f =
  let { loc_tree = t; loc_ctx = c } = l in
  { loc_tree = (f t); loc_ctx = c }

(** val distanceToTop_aux : context -> nat -> nat **)

let rec distanceToTop_aux c x =
  match c with
    | Coq_top -> x
    | Coq_leftC (d, t) -> distanceToTop_aux d (plus x (S O))
    | Coq_rightC (e, t) -> distanceToTop_aux e (plus x (S O))
```

```
 (** val distanceToTop : location -> nat **)

 let distanceToTop l =
   let { loc_tree = loc_tree0; loc_ctx = c } = l in distanceToTop_aux c O

 (** val moveToTop_aux : nat -> location -> location **)

 let rec moveToTop_aux x l =
   match x with
     | O -> l
     | S n -> moveToTop_aux n (moveUp l)

 (** val moveToTop : location -> location **)

 let moveToTop l =
   moveToTop_aux (distanceToTop l) l

 (** val locationToTree : location -> tree **)

 let locationToTree l =
   (moveToTop l).loc_tree


end
```