# Monadic, Prompt Lazy Assertions in Haskell[⋆]

Olaf Chitil[1] and Frank Huch[2]

[1] University of Kent, UK
[2] University of Kiel, Germany

**Abstract.** Assertions test expected properties of run-time values without disrupting the normal computation of a program. We present a library for enriching Haskell programs with assertions. Expected properties can be specified in a parser-combinator like language. The assertions are lazy: they do not force evaluation but only examine what is evaluated by the program. They are also prompt: assertion failure is reported as early as possible. The implementation is based on lazy observations and continuation-based coroutines.

## 1 Introduction

Assertions are parts of a program that, instead of contributing to the functionality of the program, express properties of run-time values the programmer expects to hold. It has long been recognised that augmenting programs with assertions improves software quality. An assertion both documents an expected property (e.g. a pre-condition, a post-condition, an invariant) and tests this property at run-time. For example, an assertion may express that the argument of a square root function has to be positive or zero and likewise the result is positive or zero. Assertions can be an attractive alternative to unit tests. Assertions simplify the task of locating the cause of a program fault: in a computation faulty values may be propagated for a long time until they cause an observable error, but assertions can detect such faulty values much earlier.

We can easily define a combinator for attaching assertions to expressions:

```
assert :: Bool -> a -> a
assert b x = if b then x else error "Assertion failed."
```

The assertion is an identity function when the expected property holds, but raises an exception otherwise[3]. Then, assertions can be defined as normal Haskell functions to express expected properties, for example

```
ordered :: Ord a => [a] -> Bool
ordered []       = True
ordered [_]      = True
ordered (x:y:ys) = x<y && ordered (y:ys)
```

[⋆] This work has been partially supported by the German Research Council (DFG) under grant Ha 2457/5-2 and by the United Kingdom under EPSRC grant EP/C516605/1.

[3] The Glasgow Haskell Compiler provides a variant that produces a more informative error message that includes the source location of the failed assert call.

and use them to assert for example a pre-condition:

```
checkedInsert :: Ord a => a -> [a] -> [a]
checkedInsert x xs = assert (ordered xs) (insert x xs)

insert :: Ord a => a -> [a] -> [a]
insert x [] = [x]
insert x (y:ys) = if x < y then x:y:ys else y : insert x ys
```

In many applications such assertions work fine

```
> checkedInsert 4 [1,3,2,5]
Assertion failed.
```

but sometimes they do not, as the following non-terminating expression shows:

```
> take 4 (checkedInsert 4 [1,2..])
```

In our example the function `ordered`, which expresses our expected property, is fully strict and thus forces evaluation of the whole infinite list. Programming with assertions as above results in strict programs and thus a loss of the expressive power of laziness, for example, the use of infinite data structures and cyclic definitions. As long as an assertion does not fail, a program augmented with assertions should have exactly the same input/output behaviour as the one without assertions. Hence assertions for a lazy language should be lazy, that is, a property should only be checked for the part of a data structure that is evaluated during the computation anyway.

Our example above also demonstrates that using Boolean functions for specifying properties is rather limiting in expressiveness. We want to say that any list containing two neighbouring elements in the wrong order should raise an assertion failure, also when most of the rest of the list has not been evaluated. However, `ordered` only decides on totally evaluated finite lists. We present a parser-combinator like monadic language for expressive lazy assertions. Parser combinators are a well-known tool for describing a set of token sequences. Similarly our assertion combinators describe a set of possibly partial expected values.

Whenever a part of a value is evaluated that violates an asserted property, the assertion immediately fails. We say our assertions are *prompt*. Promptness ensures that the reported unexpected value is as unevaluated as possible and thus smaller to read. Furthermore, a program fault usually violates many assertions, but promptness ensures that the assertion that is closest to the fault with respect to data flow is reported. In summary, our assertions have the following properties:

– Lazy: They do not modify the lazy behaviour of a program.
– Prompt: The violation of an assertion is reported as early as possible, before a faulty value is used by the surrounding computation.
– Expressive: Complex properties can be expressed using full Haskell.
– Portable: Assertions are implemented as a library and do not need any compiler or run-time modifications; the only extension to Haskell 98 used for the implementation are `unsafePerformIO` and `IORef`s.

## 2 Using the Assertion Monad

Expected properties are specified in an assertion monad `Try a` that combines pattern matching and non-deterministic computations. The combinators are used very similarly to standard monadic parser combinators [8].

Here is a specification of the ordered property from the Introduction:

```
ordered :: Ord a => Lazy [a] -> Try ()
ordered xs = pNil xs
        ||| (do (_,ys) <- pCons xs; pNil ys)
        ||| (do (x,ys) <- pCons xs; (y,_) <- pCons ys;
                ((do rx <- pVal x; ry <- pVal y; guard (rx < ry))
                 &&& ordered ys))
```

The tested argument is wrapped within a new type constructor `Lazy` and the result type has to be `Try ()`. Together these two types enable prompt and lazy evaluation of assertions. To specify the three different cases for lists of length zero, one, and longer lists, the assertion monad `Try a` provides the non-deterministic choice operator `(|||) :: Try a -> Try a -> Try a`. For a fair evaluation, that is, there is no fixed order in which the different cases are evaluated. Similarly, we provide a fair, parallel[4] conjunction operator `(&&&) :: Try () -> Try () -> Try ()`, which here allows independent testing at every position within the list.

For pattern matching we provide the following pattern combinators within the assertion monad:

```
pNil :: Lazy [a] -> Try ()
pCons :: Lazy [a] -> Try (Lazy a,Lazy [a])
pVal :: Lazy a -> Try a
```

For each data constructor we provide a pattern combinator that matches only the constructor and that yields the sub-structure as a tuple within the `Try` monad. For example, for the empty list it returns the empty tuple and for `(:)` it returns a pair consisting of the element and the remaining list. The combinator `pVal` matches every value and directly corresponds to a variable in a Haskell pattern. Finally, the function `guard` is the standard Haskell function that integrates a Boolean test into a `MonadPlus`.

To attach an assertion to an expression we provide the function `assert :: String -> (Lazy a -> Try ()) -> a -> a`. The first parameter is a label naming the assertion. When an assertion fails, the computation aborts with an appropriate message that includes the assertion's label. As further parameters `assert` takes the property and the value on which it behaves as a partial identity.

For expected values an assertion is an identity function. For partial values that are smaller than expected values (in the standard ordering where unevaluated/undefined is less than any value) the assertion cannot be decided and hence it is also the identity function. For any unexpected value the assertion raises an exception.

[4] That is, it has no fixed sequential evaluation order for the two arguments.

To prevent an assertion from evaluating too much, the property has to be defined as a predicate on the tested data structure. The implementation of `assert` uses a class `Observe` to ensure that only the context in which the application of `assert` appears determines how far the tested data structure is evaluated and only that part is passed to the predicate.

```
insertWithPre :: (Ord a,Observe a) => a -> [a] -> [a]
insertWithPre x xs = insert x (assert "insert input ordered" ordered xs)
```

The assertion is evaluated in a prompt, lazy manner, as the following call shows:

```
> take 4 (insertWithPre 4 ([3,4] ++ [1,2..]))
[3,4,
Assertion (insert input ordered) failed: 3 :4:1: _
```

Beside reporting the failed assertion, we also present the wrong value to the user and highlight those parts that contribute to the failure. Here these are, beside the unordered values, all `(:)` constructors above the unordered values, because the assertion would not have failed if any of them was `[]`.

Similar to this precondition, we can add a postcondition specifying that the result of `insert` is ordered. However, this is not exactly what one would like to specify as a property of `insert`. In case `insert` is called with an unordered list, this fault should not be blamed on `insert`, but on the function applying `insert` to an unordered list. A better specification for `insert` is: if the argument list is ordered, then the result is ordered as well. In contrast to the first assertion, this property is defined for a function. It specifies properties for an argument and the result. Functional assertions can be expressed by means of function `fun`$n$ [5] for functions of arity $n$:

```
insertChecked :: (Ord a, Observe a) => a -> [a] -> [a]
insertChecked = assert "insert preserves ordered property"
                       (fun2 (\ _ ys zs -> ordered ys ==> ordered zs))
                       insert
```

To express the dependence between the two ordered properties, we can use an implication (simply defined as `x ==> y = notAssert x ||| y`). Executing `insertChecked` yields the following behaviour:

```
> insertChecked 3 [5,3,4]
[5,3,3,4]
> insertChecked 3 [2,3,4]
[2,3,
Assertion (insert preserves ordered property) failed:
3  -> ( 2:3:4:[] -> 2 :3:3: 4:_)
```

In the second case highlighting shows that for the ordered input list [2,3,4] the duplicate occurrence of `3` in the result list does not meet the specification. To correct the program, we could omit duplicated elements.

[5] `fun2 :: (Lazy a -> Lazy b -> Lazy c -> Try ()) -> Lazy (a->b->c) -> Try ()`

## 3 The Idea of Respecting Laziness

This section outlines how the types `Try a` and `Lazy a` enable Haskell computations to respect how far arguments are evaluated. We introduce the data type

```
data EvalTree = Eval [EvalTree] | Uneval
```

An `EvalTree` represents how far a corresponding data structure is evaluated. It has the same tree structure as the data structure itself except that parts may be cut off by the constructor `Uneval`; that is, if the data structure contains an $n$-ary evaluated constructor, then the corresponding `EvalTree` contains an `Eval` node with $n$ `EvalTrees` in the argument list. For instance, the evaluation of list `[1,2,3]` in the call of `[1,2,3]!!1` is represented by the `EvalTree`: `Eval [Uneval,Eval [Eval [],Uneval]]`. In later sections we will refine the definition of `EvalTree` further. Now we can introduce the type synonym

```
type Lazy a = (EvalTree,a)
```

in which values are paired with their corresponding evaluation information. Because in Haskell pattern matching works from left to right, some of our later definitions are simplified by having the evaluation information as first component of the pair. The `Lazy a` type enables us to define an assertion that respects the evaluation state of the tested value, for example a function `checkOrdered` that checks whether a given list is ordered with respect to its evaluated parts:

```
checkOrdered :: Lazy [Int] -> Maybe Bool
checkOrdered (Eval [], [])           = Just True
checkOrdered (Eval [_,Eval []], [_]) = Just True
checkOrdered (Eval [eX,eYXs@(Eval [eY,eXs])], (x:yxs@(y:xs))) =
                         leq (eX,x) (eY,y) &|& checkOrdered (eYXs,yxs)
checkOrdered _                       = Nothing

leq :: Lazy Int -> Lazy Int -> Maybe Bool
leq (Eval [],x) (Eval [],y) = Just (x <= y)
leq _           _           = Nothing

(&|&) :: Maybe Bool -> Maybe Bool -> Maybe Bool
(Just True)  &|& (Just True)  = Just True
(Just False) &|& _            = Just False
_            &|& (Just False) = Just False
_            &|& _            = Nothing
```

The result type of `checkOrdered` reflects that besides being ordered or not, there is a third alternative (`Nothing`), namely that at this stage of evaluation it is not possible to decide whether the list is ordered or not. For comparing two elements of the list we use a variation of `(<=)` that also respects the `EvalTree`. Finally, the results of each comparison of two elements are combined by a modified version of `(&&)`. Besides using the extended type `Maybe Bool` this function also implements a parallel version of `(&&)` by means of its third rule. Independent of the other argument, `(&|&)` propagates an argument `Just False` as a result.

How can this approach be generalised to arbitrary computations on lazy values? Although, assertions have to return Boolean values as result, subcomputations may return other result types. Here we can also use the `Maybe a` type to express that we either obtain a result of type `a` or have a suspension.

## 4 Non-Determinism

Looking ahead, we do want to restart suspensions when more parts of a tested data structure have been evaluated. Hence we need to keep track of all separate suspensions and cannot simply conflate several into one (`Nothing &|& Nothing = Nothing`). The solution is to use a list of `Maybe` values as result for computations on lazy values. Each individual result may not be computable because of insufficient evaluation.

```
newtype Try a = Try [Maybe a]

failT = Try []
suspT = Try [Nothing]
```

The type constructor `Try` forms a monad, namely the standard combination of the non-determinism list monad and the Maybe monad, in which functions are applied to all list elements.

```
instance Monad Try where
  (Try as) >>= f = Try $ concatMap (applyRes (fromTry . f)) as
    where fromTry (Try x) = x
          applyRes :: (a -> [Maybe b]) -> Maybe a -> [Maybe b]
          applyRes f (Just x) = f x
          applyRes f Nothing = [Nothing]

  return x = Try [Just x]
```

For non-deterministic branching we define a parallel disjunction operator, which collects all possible results[6]:

```
(|||) :: Try a -> Try a -> Try a
(Try xs) ||| (Try ys)  = Try (xs++ys)
```

Within the `Try` monad we can now define pattern combinators for matching lazy values. For example:

```
pCons :: Lazy [a] -> Try (Lazy a,Lazy [a])
pCons (Eval [eX,eY],(x:xs)) = return ((eX,x),(eY,xs))
pCons (Eval _,_) = failT
pCons (Uneval,_) = suspT

pNil :: Lazy [a] -> Try ()
pNil (Eval _,v) = if null v then return () else failT
pNil (Uneval,_) = suspT
```

[6] In fact, `Try` can also be made an instance of `MonadPlus` with `mplus = (|||)` and `mzero = failT`.

These pattern combinators respect the evaluation of a given argument. If the argument is not evaluated at all, then the result is a suspension. If the constructor is evaluated and it is the wrong constructor, then matching fails. Finally, if the constructor matches, then we succeed and return the sub-terms together with their evaluation information. Similarly we define a pattern combinator that strictly matches any value.

```
pVal :: Lazy a -> Try a
pVal (et,v) = condEval et (return v)

condEval :: EvalTree -> a -> a
condEval (Eval ets) tv = foldr condEval tv ets
condEval Uneval _ = suspT
```

The combinator `pVal` is mostly used for flat data types such as `Int` or `Char`.

Next we define the parallel `(&&)` function within our framework. We start with a more general function, which applies arbitrary result functions to `Try` results:

```
(***) :: Try (a -> b) -> Try a -> Try b
(***) (Try fs) (Try xs) = Try [res | fRes <- fs, xRes <- xs,
                                     let res = do f <- fRes
                                                  x <- xRes
                                                  return (f x)]
type Assert = Try ()

(&&&) :: Assert -> Assert -> Assert
t1 &&& t2 = (return (\x1 x2 -> ()) *** t1) *** t2
```

Whereas our old `(&|&)` on type `Maybe Bool` could produce only one of three values, the new `(&&&)` may produce a value representing many successful and suspended computations.

Now it is possible to define the `ordered` assertion from Section 2. For a complete implementation it remains to show how the `EvalTree` can successively be constructed during the computation.

## 5 Generating EvalTrees

To generate evaluation information for data structures we use the idea of *observations*, first introduced by Hood [6]. All values for which an assertion is specified are *observed*. An observation constructs a corresponding `EvalTree` representing how far the data structure has been evaluated. The key idea is that the context of a computation demands head normal forms (*hnf*). Whenever such an hnf is computed we extend its `EvalTree` by means of a side effect. This means an `Uneval` leaf is replaced by `Eval [Uneval,...,Uneval]` where the number of `Uneval`s within the list is equal to the arity of the constructor of the hnf.

Because we construct and use `EvalTree`s in program parts that are not linked by data-flow and for efficiency reasons, we use mutable references (`IORef`s) in our new EvalTree representation:

```
data EvalTree = EvalR [EvalTreeRef] | UnevalR
type EvalTreeRef = IORef EvalTree
```

With this representation it is not necessary to descend into the whole data structure, when extending it in a leaf position. Instead, we can directly update the leaf.

Observable data types are represented by the following class:

```
class Observe a where
  obs :: a -> EvalTreeRef -> a
```

We demonstrate how an instance of this class can be defined by means of the list data type:

```
instance Observe a => Observe [a] where
  obs (x:xs) r = unsafePerformIO $ do [aRef,bRef] <- mkEvalTreeCons r 2
                                      return (obs x aRef : obs xs bRef)
  obs []     r = unsafePerformIO $ do mkEvalTreeCons r 0
                                      return []
```

Whenever the context demands the evaluation of an observed value, the corresponding node in the `EvalTree` is extended by means of the function

```
mkEvalTreeCons :: EvalTreeRef -> Int -> IO [EvalTreeRef]
mkEvalTreeCons r n = do refs <- sequence (replicate n emptyUnevalRef)
                        writeIORef r (EvalR refs)
                        return refs

emptyUnevalRef :: IO EvalTreeRef
emptyUnevalRef = newIORef UnevalR
```

Furthermore, observers are added to the (not yet evaluated) arguments of the resulting constructor. These observers extend on demand the `IORef`s returned by `mkEvalTree` (`aRef` and `bRef`), which are also added to the new `EvalR` node within the `EvalTree`. The initial observer can be added with the function

```
observe :: Observe a => a -> IO (EvalTreeRef,a)
observe x = do r <- emptyUnevalRef
               return (r,obs x r)
```

This function is called whenever an assertion is added to a data structure, as discussed in the next section.

On top of these functions, it is possible to define a late (in contrast to prompt) implementation of our lazy assertions. Such an implementation stores all assertions of the program within a global state. At the end of the execution, all checks within this state are executed. Failed assertions are reported to the user.

## 6 Promptness

So far, our assertions meet two major goals. They respect the laziness of the program and they provide non-determinism by means of the operators `(|||)`,

`(***)`, and `(&&&)`. However, we still want our assertions to be *prompt* for the following reasons:

- Currently substantial memory is consumed, because the assertions themselves and the underlying data structures have to be kept until the final check can be performed. The more assertions are added the more memory is needed, although some data structure is fully evaluated or the assertion can be decided already by the evaluated part. When checking assertions directly at run-time large parts of the memory would become garbage and could be reused.
- Evaluating assertions at the end of the computation means the assertion is checked on maximally evaluated data structures. If a failed assertion would be reported earlier, then smaller data structures would be presented to the user. It will often be easier to understand why an assertion was violated.
- It will often be the case that in the end not only one assertion fails. There may be many consecutive faults. But how can a user know which was the initial fault to detect the bug in the program? The order in which the assertions are printed at the end of the execution does not reflect how different assertions depend on each other. Having prompt assertions, the computation can directly stop after reporting the first violated assertion. Consecutive faults are not reported anymore.
- In non-terminating systems such as most reactive applications (e.g. a webserver or a web-browser) it is inconvenient to stop the application just for checking assertions. Users want them to be checked in parallel in the background, without effecting the run-time behavior of the program.

The implementation shall suspend checks on unevaluated parts of data structures and directly awake them when these parts are evaluated to hnf.

## 6.1 Preparing the EvalTrees

For checking an assertion many checks have to be executed concurrently on different parts of the tested data structure. Many of these checks will have to suspend, because specific parts of the data structure are not yet evaluated. We store each suspended check in the `Uneval` leaf associated with the part of the data structure that it is suspended on, so that the checks can be executed when that data part is demanded. Several checks may be associated with the same part and hence many suspended checks may have to be stored in one `Uneval` leaf. A check does not return any value (it may just raise an exception), but it reads `IORef`s to read the growing `EvalTree` and hence it is of type `IO ()`. Checks may be added to an `Uneval` leaf at different times. We simply compose all checks for one `Uneval` leaf sequentially as an `IO` action stored within an `IORef`. Arbitrary sequential composition works, because we assume that all checks terminate.

We redefine the `EvalTree` with a reference containing an `IO` action:

```
data EvalTree = Eval [EvalTreeRef] | Uneval (IORef (IO ()))
type EvalTreeRef = IORef EvalTree
```

For the construction of the new `EvalTree` only two modifications have to be made:

```
mkEvalTreeCons :: EvalTreeRef -> Int -> IO [EvalTreeRef]
mkEvalTreeCons r n = do refs <- sequence (replicate n emptyUnevalRef)
                        Uneval aRef <- readIORef r
                        action <- readIORef aRef
                        writeIORef r (Eval refs)
                        action
                        return refs

emptyUnevalRef :: IO EvalTreeRef
emptyUnevalRef = do aRef <- newIORef (return ())
                    newIORef (Uneval aRef)
```

The function `mkEvalTreeCons` is called whenever an observed expression is evaluated to hnf. Then we read the suspended assertion checks (`action`) and execute them before we return the list of new sub-references. The `emptyUnevalRef` contains an `IORef` with no action, since there is no lazy computation to be performed for that sub-value yet.

## 6.2 Coroutines

In our outline in Section 3 we defined the data type `Try` as a list of `Maybe` values. However, for implementing promptness we have to compute the assertion checks step by step whenever the `EvalTree` is extended. Hence non-determinism or coroutines through continuation passing style is a more appropriate means of implementation. We have a success continuation and a fail continuation, just like continuation-based parser combinators [9]. The success continuation must take a fail continuation as argument to support non-determinism. We already established that checks are of type `IO ()`.

```
type FailCont = IO ()
type SuccCont a = FailCont -> a -> IO ()

newtype Try a = Try (SuccCont a -> FailCont -> IO ())
```

If there exists an alternative for a failed assertion, for example by non-deterministic branching in `(|||)`, then the `SuccCont` can discard the current `FailCont`.

Now we are ready to define the `Monad` instance for the new type `Try`:

```
instance Monad Try where

  (Try asIO) >>= f =
     Try (\sc fc -> asIO (\sfc x -> fromTry (f x) sc sfc) fc)

  return x = Try (\sc fc -> sc fc x)

fromTry :: Try a -> SuccCont a -> FailCont -> IO ()
fromTry (Try x) = x
```

```
failT :: Try a
failT = Try (\sc fc -> fc)
```

Similar to constructing success continuations by means of `return`, it is handy to have a function `failT` for constructing fail continuations.

To see how this lazy `Try` monad works, we first redefine the list patterns:

```
pNil :: Lazy [a] -> Try ()
pNil (Eval _ _,[]) = return ()
pNil (Eval _ _,(_:_)) = failT
pNil rx@(Uneval ref,v) = Try (suspTIO ref (pNil rx))
```

If the data structure is already evaluated, then we either succeed or fail. If the data structure is not yet evaluated, we add a suspended computation to the corresponding `IORef` within the `EvalTree`. The action to be performed when the constructor is evaluated to hnf is the same matching again (`pNil rx`). In the definition of `suspTIO` the action within the `IORef` is extended accordingly:

```
suspTIO :: IORef (IO ()) -> Try a -> SuccCont a -> FailCont -> IO ()
suspTIO ref try c fc = do io <- readIORef ref
                          writeIORef ref (io >> (fromTry try) c fc)
```

Similarly we can define the pattern combinator `pCons`:

```
pCons :: Lazy [a] -> Try (Lazy a,Lazy [a])
pCons (Eval _ [eX,eY],(x:y)) = return ((eX,x),(eY,y))
pCons (Eval _ _,[])          = failT
pCons rx@(Uneval ref,v)      = Try (suspTIO ref (pCons rx))
```

Next we define the operator `(|||)` which allows a parallel, independent execution of two `Try` computations:

```
(|||) :: Try a -> Try a -> Try a
(Try x) ||| (Try y) = Try (\c fc -> do
  ref <- newIORef True
  x c (orIORef ref fc) >> y c (orIORef ref fc))

orIORef :: IORef Bool -> FailCont -> IO ()
orIORef ref fc = do v <- readIORef ref
                    if v then (writeIORef ref False)
                         else fc
```

Both computations have to be performed in parallel because they may compute different results of type `a`. The whole computation only fails if both subcomputations fail. For this purpose we create a synchronisation `IORef` which is set to `False` by the first failing alternative. If the other alternative fails too this alternative continues with the fail continuation `fc`. The fail continuation is passed to both alternatives, but only the second failing alternative (with respect to time, not order in the code!) executes this continuation.

We define the parallel conjunction `(&&&)` again in terms of the more general operator `(***)`:

```
(***) :: Try (a -> b) -> Try a -> Try b
(***) (Try f) (Try x) = Try $ \sc fc -> do
  fRef <- newIORef []
  xRef <- newIORef []
  ref <- newIORef True
  f (\ffc f' -> do updateIORef ((ffc,f'):) fRef
                   xs <- readIORef xRef
                   mapM_ (\(xfc,x) -> sc (ffc >> xfc) (f' x)) xs)
    (andIORef ref fc)
  x (\xfc x' -> do updateIORef ((xfc,x'):) xRef
                   fs <- readIORef fRef
                   mapM_ (\(ffc,f) -> sc (ffc >> xfc) (f x')) fs)
    (andIORef ref fc)

(&&&) :: Assert -> Assert -> Assert
t1 &&& t2 = (return (\x1 x2 -> ()) *** t1) *** t2

andIORef :: IORef Bool -> IO () -> IO ()
andIORef ref fc = do v <- readIORef ref
                     if v then writeIORef ref False >> fc
                          else return ()
```

The computation of both arguments of `(***)` may introduce non-determinism, that is, multiple values. Whenever a new value is produced within the success continuation we extend the corresponding list in `fRef` and `xRef`. Besides the different values, this list also contains the corresponding fail continuations within a pair. Furthermore, we directly apply a new function to every already computed argument in the success continuation of `f`, as well as every stored function to a new argument in the success continuation of `x`. Thus every function is applied to every argument exactly once.

If any computation fails, we update the Boolean value in `ref` to `False` and directly continue with the fail continuation. Then, if the other coroutine fails as well, the `IORef` already contains `False` and it stops immediately. Like for the disjunction operator, the fail continuation is executed at most by one coroutine.

Finally we need the definition of `assert`:

```
assert :: Observe a => String -> (Lazy a -> Assert) -> a -> a
assert label p x = unsafePerformIO (do
  (eT,x') <- observe x
  let Try check = p (eT,x)
  check (const (putStrLn ("Assertion succeeded: "++label)))
        (fail ("Assertion failed: "++label))
  return x')
```

After installing an observer for `x`, we directly start the coroutine `check` for the asserted property `p`. Usually this coroutine directly suspends itself. Its `SuccCont` ignores its current `FailCont` and simply prints that the assertion succeeded. The `FailCont` aborts the whole computation reporting the failed assertion. We have to pass `x`, not the observer-wrapped variant `x'`, to the property `p`, because a partial value of `x'` is only available after all necessary assertion checks have been performed on it. As shown in Section 2, the real implementation reports a failure with a highlighted presentation of the wrong value, which we will discuss in the next section. Furthermore, success messages are written to a file to avoid conflicts with the program output.

To provide a comfortable library, we provide some further functions on assertions, like negation, implication, and assertion variations of standard Haskell function such as `elem` and `any`.

## 7 Failure Highlighting

As presented in Section 2, the data structure violating an assertion is also presented to the user. All parts responsible for the failure are marked such that the problematic sub-structures can easily be detected. This section gives a brief overview of how this highlighting is realised in our implementation.

So far the `EvalTree` does not contain any information about the names of the constructors inside the data structure. Hence it is not possible to print data structures at all. Therefore we add a `String` parameter to the constructor `Eval` that can easily be set in `obs`. Knowing all constructor names of the observed data structure, it is straightforward to generate a string representation of the data structure containing underscores for unevaluated parts.

For syntax highlighting we have to collect some more information while checking assertions. We identify every node in the `EvalTree` with a position `Pos`:

```
data EvalTree = Eval Pos String [EvalTreeRef] | Uneval (IORef (IO ()))
```

While checking an assertion, we can then collect sets of positions (`PosSet`), representing the nodes visited during the check. We extend the success and the fail continuations with a set of positions as additional parameter:

```
type SuccCont a = PosSet -> FailCont -> a -> IO ()
type FailCont   = PosSet -> IO ()
```

When a check splits into (parallel, independent) sub-checks, both checks generate their own sets of positions which are later combined according to the branching operator. For `(|||)` both sub-checks have to fail and, hence, the sets of positions are joined. For `(***)` (e.g. `(&&&)`) only the set of positions of the failing sub-check has to be considered.

In our implementation `Pos` and `PosSet` are defined as abstract data types with functions for set manipulation. Internally, a `Pos` is implemented as a list of `Int`s where the `Int`s successively express which branch in the `EvalTree` is chosen. The set of positions is implemented as a Trie [10] over lists of `Int`s.

Although the Trie of positions has the same structure as the `EvalTree`, it is not possible to directly integrate the position information into the `EvalTree`. Many assertions are checked in parallel and for each of these parallel checks different positions have to be considered. Parts of these assertions may have succeeded and their position sets should be discarded since this part of the data structure was not responsible for the failure.

## 8 Assertions for Functions

One of the most important features of our library is the ability to assert properties of functions. This allows programmers to express pre- and postconditions as well as invariants of functions. It is possible to express arbitrary relations between arguments and results.

The basic idea of the implementation is to represent a function as its graph, as far as it is used/constructed during the program execution. At each application of the function the graph is extended with a new pair of lazily constructed values. Functional assertions usually contain checks applied to function arguments or results which have to be checked by the mechanism described so far.

In the data type `EvalTree` we add a representation for functions

```
data EvalTree = ... | Fun Pos EvalTreeRef EvalTreeRef () () EvalTreeRef
```

where the first two `EvalTreeRef`s represent the argument and result value of the (curried) function and the last `EvalTreeRef` represents the next application of the function. The two arguments of type `()` are used to store the concrete argument and result value (of arbitrary type) by means of a type-cast[7] inside the monomorphic data structure `EvalTree`. An instance of the class `Observe` is defined to construct and extend `EvalTree`s for functions.

In the `Try` monad we give access to the graph of an observed function through a function which converts the observed function into a lazily constructed (infinite) list of argument-result pairs:

```
pFun ::  Lazy (a -> b) -> Try (Lazy [(a,b)])
```

Assertions defined on this lazily constructed list are stepwise evaluated whenever the list is extended by a new function application. Functional assertions have to hold for each application. We apply them to all list elements by means of the assertion variant of the Haskell function `all`.

This conversion through a lazily constructed list is hidden in functions for a convenient construction of functional assertions of arbitrary arity:

```
fun1 :: (Lazy a -> Lazy b -> Assert) -> Lazy (a -> b) -> Assert
fun2 :: (Lazy a -> Lazy b -> Lazy c -> Assert) -> Lazy (a -> b -> c)
        -> Assert
...
```

An example for a functional assertion is presented in Section 2.

## 9 Related Work

The first systematic approach to adding assertions to a functional language targets the strict language Scheme [5]. It provides convenient constructs for expressing properties of functions, including higher-order functions, and augmenting

[7] A function `coerce :: a -> b` can be defined using the Haskell 98 extension `IORefs` in combination with `unsafePerformIO`.

function definitions with assertions. Laziness is irrelevant and promptness trivial for strict functional languages. Instead a major concern of this work is which program part to blame when an assertion fails. The approach to blaming cannot directly be transferred to a lazy language, because there the run-time stack does not reflect the call structure. Instead a cooperation with the Haskell tracer Hat [11] may provide a solution in the future. The Scheme approach has been transferred to Haskell [7], but without taking its lazy semantics into account.

The first paper on *lazy* assertions for the lazy language Haskell [2] uses normal functions with Boolean result for expressing properties and hence the assertions are not prompt. The paper gives several examples of where the lack of promptness renders the assertions useless. Furthermore, expressibility of properties of functions is limited and the implementation requires concurrency language extensions as provided only by GHC.

In the first paper on *lazy and prompt* assertions for Haskell [1] properties are expressed in a pattern logic. The logic provides quantifiers and context patterns that allow referring to substructures of the tested value. However, most Haskell users find this logic hard to understand and many simple properties, such as that two lists have the same lengths, require complex descriptions. The implementation of the pattern logic is only sketched.

QuickCheck is a library for testing Haskell functions with random data [3]. Normal Boolean functions express expected properties, for example

```
prop :: Int -> [Int] -> Property
prop x xs = ordered xs ==> ordered (insert x xs)
```

where `ordered :: [Int] -> Bool` states that the function `insert` preserves order. Normal Boolean functions can be used, because only total, finite data structures are tested. An extension for (finite) partial values [4] has fundamental limits whereas our assertions fully support laziness. It can be very hard to generate random test data, for example input strings for a parser that are likely to be parseable. QuickCheck can only test top-level functions whereas an assertion can be attached to any local definition or subexpression. So testing with random data and testing with real data as our assertions do are two different methods which complement each other.

## 10 Conclusions

We have presented a new approach to augmenting lazy functional programs such as Haskell with assertions. The implementation is based on a technically interesting combination of continuation-based non-determinism, explicit scheduling of concurrent processes and HOOD-like observation of values. However, it is a portable library that requires only two common extensions of Haskell 98, `unsafePerformIO` and `IORef`s, which are supported by all Haskell compilers. The assertions are lazy and prompt. Most importantly, the combinator language for expressing asserted properties is easy to use, because it is similar to familiar parser combinator libraries. It combines pattern matching and non-deterministic computations. Furthermore, it is very expressive, allowing the formulation of any imaginable computable property. Assertions for functional values are easy to write and syntax highlighting simplifies the identification of parts of a value that are relevant for a failure.

The library does not prevent the user from writing assertions that change the program semantics by causing non-termination or raising an exception; after all, an asserted property may evaluate any Haskell expression, including `undef` or `error`. However, the library enables the user to formulate complex properties for partial and infinite values.

In the future we intend to investigate a theoretical formalisation of our assertions and to import ideas from strict assertions for Haskell [7].

## References

1. Olaf Chitil and Frank Huch. A pattern logic for prompt lazy assertions in Haskell. In Andrew Butterfield Zoltan Horvath, editor, *Implementation and Application of Functional Languages: 18th International Workshop, IFL 2006*, volume 4449 of *LNCS*. Springer, 2007.
2. Olaf Chitil, Dan McNeill, and Colin Runciman. Lazy assertions. In Phil Trinder, Greg Michaelson, and Ricardo Pena, editors, *Implementation of Functional Languages: 15th International Workshop, IFL 2003*, LNCS 3145, pages 1–19. Springer, November 2004.
3. K. Claessen and R. J. M. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proc. 5th Intl. ACM Conference on Functional Programming*, pages 268–279. ACM Press, 2000.
4. Nils Anders Danielsson and Patrik Jansson. Chasing bottoms, a case study in program verification in the presence of partial and infinite values. In Dexter Kozen, editor, *Proceedings of the 7th International Conference on Mathematics of Program Construction, MPC 2004*, LNCS 3125, pages 85–109. Springer-Verlag, July 2004.
5. Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 48–59. ACM Press, 2002.
6. A. Gill. Debugging Haskell by observing intermediate datastructures. *Electronic Notes in Theoretical Computer Science*, 41(1), 2001. (Proc. 2000 ACM SIGPLAN Haskell Workshop).
7. Ralf Hinze, Johan Jeuring, and Andres Löh. Typed contracts for functional programming. In *Proceedings of the 8th International Symposium on Functional and Logic Programming, FLOPS 2006*, LNCS 3945, pages 208–225, 2006.
8. Graham Hutton and Erik Meijer. Monadic parsing in Haskell. *J. Funct. Program.*, 8(4):437–444, 1998.
9. Chris Okasaki. Functional pearl: Even higher-order functions for parsing or Why would anyone ever want to use a sixth-order function? *Journal of Functional Programming*, 8(2):195–199, 1998.
10. Chris Okasaki. *Purely Functional Data Structures.* Cambridge University Press, Cambridge, UK, 1998.
11. M. Wallace, O. Chitil, T. Brehm, and C. Runciman. Multiple-view tracing for Haskell: a new Hat. In *ACM Workshop on Haskell*, 2001.