# Lazy Assertions
## — Draft —

Olaf Chitil, Dan McNeill and Colin Runciman

Department of Computer Science, The University of York, UK

**Abstract.** Assertions test expected properties of run-time values without dissrupting the normal working of a program. So in a lazy functional language assertions should be lazy — not forcing evaluation, but only examining what is evaluated by other parts of the program. We describe two different ways of embedding lazy assertions in Haskell, one sequential and the other concurrent. Examples illustrate the relative merits of the two approaches. We also show that timely failure of lazy assertions may require assertions in assertions!

## 1 Introduction

A programmer writing a section of code typically does so with many assumptions or intentions about the values involved. Some of these assumptions or intentions are expressed in a way that can be verified by a compiler, for example as part of a type system. But many are beyond the expressive power of types amenable to automatic static checking. It may be possible to formulate some of these deeper properties as theorems to be proved in a suitable logic, but as this option involves an expensive and highly specialised activity it is rarely pursued.

Instead of leaving essential properties unexpressed and unchecked, a useful and comparatively simple option is to express them as *assertions* — boolean-valued expressions that the programmer assumes or intends will always be true. Assertions are checked at run-time as they are encountered, and any failures are reported. In the absence of any such failure, the program runs just as it would without any assertions, apart from the extra time and space needed for checking.

The usefulness of assertions in conventional state-based programming has long been recognised, and many imperative programming systems include some support for them. In these systems, each assertion is attached to a *program point*; whenever control reaches that point the corresponding assertion is immediately evaluated to a boolean result. Important special cases of program points with assertions include points of entry to, or return from, a procedure with *preconditions* or *postconditions*.

In a functional language, the basic units of programs are expressions rather than commands. The commonest form of expression is a function application. So our first thought might be that an assertion in a functional language can simply be attached to an expression: an assertion about arguments (or 'inputs') alone can be checked before the expression is evaluated and an assertion involving the result (or 'output') can be checked afterwards. But in a lazy language this view is at odds with the need to preserve normal semantics. Arguments may be unevaluated when the expression is entered, and may remain unevaluated or only partially evaluated even after the expression has been reduced to a result. The result itself may only be evaluated to *weak head-normal form*. So neither arguments nor result can safely be the subjects of an arbitrary boolean assertion that could demand their evaluation in full.

Here is the problem we address in this paper. How can assertions be introduced in a lazy functional language? How can we satisfy our eagerness to evaluate assertions, so that failures can be caught as soon as possible, without compromising the lazy evaluation order of the underlying program to which assertions have been added?

Our aim is an *embedded* solution to this problem. That is, we aim to support assertions by a small but sufficient library defined in the programming language itself. This approach avoids the need to modify compilers or run-time systems and gives the programmer a straightforward and familiar way of using a new facility. Specifically, we shall be programming in Haskell[3].

The rest of the paper is organised as follows. Section 2 uses two examples to illustrate the problem with eager assertions in a lazy language. Section 3 outlines and illustrates the contrasting nature of lazy assertions. Section 4 describes an implementation of lazy assertions that postpones their evaluation until the underlying program is finished. Section 5 describes an alternative implementation in which each assertion is evaluated by a concurrent thread. Section 6 addresses a residual problem of sequential demand within assertions. Section 7 discusses related work. Section 8 concludes and suggests future work.

## 2 Eager Assertions Must be True

The library provided with the GHC compiler already includes a function `assert :: Bool -> a -> a`. It is defined in such a way that `assert True x = x` but an application of `assert False` causes execution to halt with a suitable error message. An application of `assert` always expresses an *eager* assertion because it is a strict function: evaluation is driven by the need to reduce the boolean argument, and no other computation takes place until the value `True` is obtained.

To explore the consequences of this eager definition of assert we shall look at two examples. We return to the same examples in later sections.

### Example 1: sets represented as ordered trees

Consider the following datatype.

```
data Ord a => Set a  = Empty
                     | Union (Set a) a (Set a)
```

Functions defined over sets include `with` and `elem`, where ``s `with` x`` represents $s \cup \{x\}$ and ``x `elem` s`` represents the membership test $x \in s$.

```
with :: Ord a => Set a -> a -> Set a
Empty           `with` x = Union Empty x Empty
(Union s1 y s2) `with` x = case compare x y of
                           LT -> Union (s1 `with` x) y s2
                           EQ -> Union s1 y s2
                           GT -> Union s1 y (s2 `with` x)

elem :: Ord a => a -> Set a -> Bool
x `elem` Empty           = False
x `elem` (Union s1 y s2) = case compare x y of
                           LT -> x `elem` s1
                           EQ -> True
                           GT -> x `elem` s2
```

The `Ord a` qualification in the definition of `Set` and in the signatures for `with` and `elem` only says that comparison operators are defined for the type `a`. It does *not* guarantee that `Set a` values are strictly ordered trees as the programmer intends. To assert this property, we could define the following predicate.

```
strictlyOrdered :: Ord a => Set a -> Bool
strictlyOrdered = soBetween Nothing Nothing
    where
    soBetween _  _  Empty           = True
    soBetween lo hi (Union s1 x s2) = between lo hi x &&
                                      soBetween lo (Just x) s1 &&
                                      soBetween (Just x) hi s2
    between lo hi x = maybe True (< x) lo && maybe True (> x) hi
```

Something else the programmer intends is a connection between `with` and `elem`. It can be expressed by asserting ``x `elem` (s `with` x)``. Combining this property with the ordering assertion we might define:

```
s `checkedWith` x = assert post s'
                    where
                    s'   = assert pre s `with` x
                    pre  = strictlyOrdered s
                    post = strictlyOrdered s' && x `elem` s'
```

*Observations* The eager assertions in `checkedWith` may 'run ahead' of evaluation actually required by the underlying program, forcing fuller evaluation of tree structures and elements. The strict-ordering test is a conjunction of two comparisons for *every* internal node of a tree, forcing the entire tree to be evaluated (unless the test fails). Even the check involving `elem` forces the path from the root to `x`.

Does this matter? Surely some extra evaluation is inevitable when non-trivial assertions are introduced? It could even have the helpful side-effect of fixing a space-leak! It does matter, it is not inevitable (as later sections show) and the

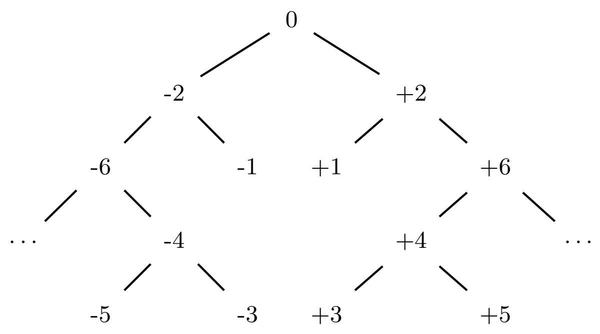

**Fig. 1.** A tree representation of the infinite set of integers. Each integer $i$ occurs at a depth no greater than $2\log_2(\text{abs}(i) + 1)$. Differences between adjacent elements on leftmost and rightmost paths are succesive powers of two.

consequences may be far from helpful. Forcing evaluation can just as easily cause space leaks as cure them [8]. It can also cause assertion-checking to degenerate into a pre-emptive, non-terminating and unproductive process. What if, for example, a computation involves the set of all integers, represented as in Figure 1? Functions such as `elem` and `with` still produce useful results. But `checkedWith` eagerly carries the whole computation away on an infinite side-track!

### Example 2: the infinite sequence of primes

A standard exercise in programming with lazy lists is to enumerate the primes. Here is one solution:

```
primes = sieve [2..]

sieve (p:xs) = p : filter (noFactorIn primes) xs

noFactorIn (p:ps) x = p*p > x ||
                      x `mod` p > 0 && noFactorIn ps xs
```

If we consider only the assertion that `primes` is an ordered sequence, we would merely revisit the lessons from the previous example: a famous property of the primes is that there are infinitely many of them. But what if we wish to assert this very property for the programmed `prime` sequence? Using eager `assert`, no single assertion can be formulated. We cannot possibly compute the entire sequence of primes, verify that it is truly infinite, and then continue! Rather, to express the expectation that a list is infinite requires infinitely many assertions that each recursive tail is non-empty. Using the definition

```
infinitelyMany xs =
  assert (not null xs) (head x : infinitelyMany (tail xs))
```

we can evaluate `infinitelyMany primes`. A similar technique of interleaving atomic assertions with a copying process could be used to handle the infinite ordered trees in the previous example. This is left as an exercise for the reader — but see Section 6.

## 3 Lazy Assertions Must Not Be False

We have seen that the need for laziness in the evaluation of assertions may be inherent in the nature of the program, or in the nature of the assertion, or both. Laziness in this context means that the evaluation of assertions should only examine those parts of their subject data structures that are in any case demanded by the underlying program. Lazy assertions should make a (provisional) assumption of validity about other data not (yet) evaluated. Computation of the underlying program should proceed not only if an assertion reduces to `True`, but also if it cannot (yet) be reduced to a value at all; the only constraint is that an assertion must never reduce to `False`.

If we are to guard data structures that are the subjects of assertions from over-evaluation, we cannot continue to allow arbitrary boolean expressions involving these structures. We need to separate the *predicate* of the assertion from the *subject* to which it is applied. An implementation of assertions should combine the two using only a special evaluation-safe form of application. So the type of `assert` becomes

```
assert :: (a -> Bool) -> a -> a
```

where `assert p` acts as a lazy partial identity.

If we had an implementation of this lazy `assert`, how would it alter the examples we looked at before?

### Example 1 revisited

In view of the revised type of `assert`, the definition of `checkedWith` must be altered slightly, making `pre` and `post` predicates rather than booleans.

```
s ‘checkedWith‘ x = assert post (assert pre s ‘with‘ x)
                    where
                    pre  = strictlyOrdered
                    post = \s’ -> strictlyOrdered s’ && x ‘elem‘ s’
```

Now the computation of `checkedWith` applications should progress just like an unchecked application of `with`. If infinite sets are involved, the corresponding assertions are only partially computed, up to the limits imposed by the finite needed parts of these sets.

**Example 2 revisited**

In a lazy language, the wish to write a single assertion that a list is infinite seems only reasonable. Given a lazy `assert` we can use a predicate `infinitelyMany` defined like this

```
infinitelyMany [] = False
infinitelyMany xs = infinitelyMany (tail xs)
```

with `primes` as the subject. The single-assertion computation

```
assert infinitelyMany primes
```

should behave just like the computation of `primes` alone.

## 4 Sequential Implementation

Now we see how we can implement a library for lazy assertions in Haskell. We develop the library in steps: we give a working version, criticise it, and then refine it to the next version. This section concludes with a sequential implementation that is usable but has some disadvantages.

### 4.1 Delayed Assertions

First we want to ensure that the evaluation of the assertions cannot disturb the evaluation of the underlying program. We do so by evaluating all assertions *after* termination of the main computation. The main computation only evaluates the underlying program and collects all assertions.

The implementation uses some extensions of the Haskell 98 standard: Extended exceptions enable a program to catch all erroneous behaviour of a sub-computation , `IORef`s add mutable variables to the IO monad, and the function `unsafePerformIO :: IO a -> a` enables us to implement `assert` using exceptions and mutable variables without giving it a monadic type [7].

We introduce a global mutable variable `finalisers` that stores a list of pending assertions, to be checked at the end of the main computation.

```
finalisers :: IORef [IO ()]
finalisers = unsafePerformIO $ newIORef []
```

The function `assert` simply adds an assertion to the `finalisers` list. The function also takes a string as argument to simplify identification when an assertion fails. Only evaluation of the action `evalAssertion n p x` actually evaluates the assertion of name `n` and predicate `p` with test argument `x`. The function `evalAssert` has to catch exceptions to ensure that an exception in one assertion does not prevent the remaining pending assertions from being tested.

```
assert :: String -> (a -> Bool) -> a -> a
assert n p x = unsafePerformIO $ do
  fins <- readIORef finalisers
  writeIORef finalisers (evalAssertion n p x : fins)
  return x

evalAssertion :: String -> (a -> Bool) -> a -> IO ()
evalAssertion n p x = do
  Control.Exception.catch
    (when (not (p x))
       (hPutStrLn stderr ("\nAssertion " ++ show n ++ " failed.")))
    (\e -> hPutStrLn stderr
             ("\nAssertion " ++ show n ++
              " raised exception: " ++ show e)
```

To use assertions we have to wrap the action corresponding to the underlying program by applying `runA` to it. To ensure that the assertions are always run at the end of the computation, the definition of `runA` has to catch any exception occurring in the main computation.[1]

```
runA :: IO a -> IO ()
runA io = do
  Control.Exception.catch io
    (const (putStrLn "Exception occurred in main computation" >>
            return undefined))
  fins <- readIORef finalisers
  sequence_ fins
```

*Properties of the Implementation.* This simple implementation does not prevent an assertion from evaluating a test argument further than the main computation itself. Because assertion checking is delayed, over-evaluation cannot disturb the main computation, but it can cause run-time errors or non-termination in the evaluation of an assertion (see Section 2).

## 4.2 Avoiding Over-Evaluating

To avoid over-evaluation do we need any non-portable "function" for testing if an expression is evaluated? No, exceptions and the function `unsafePerformIO` are enough. We can borrow and extend a technique from the Haskell Object Observation Debugger (HOOD) [4]. We arrange that as evaluation of the underlying program demands the value of an expression wrapped with an assertion, the main computation makes a copy of the value. Thus the copy comprises exactly

[1] The variable `finalisers` is initialised with the empty list. However, interactive interpreters may not reevaluate a CAF such as `finalisers` every time a new expression is interactively evaluated. Hence to ensure correct initialisation we have to insert `writeIORef finalisers []` as first line in the `do` block of `runA`.

those parts of the value that were demanded by the evaluation of the underlying program.

We introduce two new functions, `demand` and `listen`. The function `demand` is wrapped around the value that is consumed by the main computation. The function returns that value and, whenever a part of the value is demanded, the function also adds the demanded part to the copy. The function `listen` simply returns the copy; because `listen` is only evaluated after the main computation has terminated, `listen` returns those parts of the value that were demanded by the main computation. If the result of `listen` is evaluated further, than it raises an exception. For every part of a value there is a `demand / listen` pair that communicates via an `IORef`. The value of the `IORef` is `Unblocked v` to pass a value (weak head normal form) or `Blocked` to indicate that the value was not (yet) demanded. The implementation of `demand` is specific for every type. Hence we introduce a class and here we give only one exemplary instance for lists. We discuss in an appendix how to reduce the effort of writing these instances.

```
data ValState a = Blocked | Unblocked a

class Assert a where
  demand :: IORef (ValState a) -> a -> a

instance Assert a => Assert [a] where
  demand r [] = unsafePerformIO $ do
    writeIORef r (Unblocked [])
    return []
  demand r (x:xs) = unsafePerformIO $ do
    r1 <- newIORef Blocked
    r2 <- newIORef Blocked
    writeIORef r (Unblocked (listen r1 : listen r2))
    return (demand r1 x : demand r2 xs)

listen :: IORef (ValState a) -> a
listen r = unsafePerformIO $ do
  val <- readIORef r
  case val of
    Blocked -> error "blocked"
    Unblocked x -> return x
```

We have to adapt our implementation of `assert` to use `demand` and `listen`

```
assert :: Assert a => String -> (a -> Bool) -> a -> a
assert s p x = unsafePerformIO $ do
  r <- newIORef Blocked
  fins <- readIORef finalisers
  writeIORef finalisers (evalAssertion s p (listen r) : fins)
  return (demand r x)
```

Finally the evaluation of an assertion has to handle the case that it is blocked to avoid over-evaluation:

```
evalAssertion :: String -> (a -> Bool) -> a -> IO ()
evalAssertion n p x = do
  Control.Exception.catch
    (when (not (p x))
      (hPutStrLn stderr ("\nAssertion " ++ show n ++ " failed.")))
    (\e -> case e of
             ErrorCall "blocked" -> return ()
             _ -> hPutStrLn stderr ("\nAssertion " ++ show n ++
                                    " raised exception: " ++
                                    show e))
```

*Properties of the Implementation.* An assertion can use exactly those parts of values that are evaluated by the main computation, no less, no more. However, if an assertion fails, the programmer is informed rather late; because of the problem actually detected by the assertion, the main computation may have run into a run-time error or worse a loop. The computation is then also likely to produce a long, fortunately ordered, list of failed assertions. A programmer wants to know about a failed assertion before the main computation uses the faulty value! Additionally, both this and the previous implementation retain all tested values until the end of the computation, so that most realistic computations will run into space performance problems.

## 5 Concurrent Implementation

How can we evaluate assertions as eagerly as possible yet still only using data that is demanded by the main computation? Rather than delaying assertion checking to the end, we can evaluate each assertion in a separate thread concurrently to the main computation. When an assertion demands a part of a value that has not yet been demanded by the main computation, the assertion thread is blocked and control is passed to the main thread. Whenever the main thread demands another part of the tested value and an assertion thread is waiting for that value, the main thread is blocked and control is passed to the assertion thread. Thus the assertion always gets a new part of the value for testing before it is used by the main computation. Coroutining is used to pass control between an assertion thread and the main thread.

So this implementation requires a further extension: Concurrent Haskell [7].The function `forkIO` starts a new thread. We also use the quantity semaphore type `QSem`. The functions `waitQSem` blocks a thread until a ‘unit’ of a semaphore becomes available, and `signalQSem` makes a ‘unit’ available.

To control the running status of a pair of threads we introduce a `Switch` of two binary semaphores and associated functions for passing control.

```
data Switch = S QSem QSem

initSwitch :: IO Switch
initSwitch = do mainS <- newQSem (-1)
                assertS <- newQSem (-1)
                return (S mainS assertS)

continueAssert :: Switch -> IO ()
continueAssert (S mainS assertS) = do signalQSem assertS
                                      waitQSem mainS

continueMain :: Switch -> IO ()
continueMain (S mainS assertS) = do signalQSem mainS
                                    waitQSem assertS

finishAssert :: Switch -> IO ()
finishAssert (S mainS _) = signalQSem mainS
```

A part of a tested value can be in any of three states: (1) not yet demanded by either the main or the assertion thread, (2) demanded by the assertion thread which is hence blocked, and (3) evaluated, because it was demanded by the main thread:

```
data ValState a = Untouched | DemandedByAssert | Evaluated a
```

The basic idea of copying the test value on demand is still the same as before. As a helper for the function `demand` we introduce the function `copy`. It distinguishes the states `DemandedByAssert` and `Evaluated` and passes control to the assertion thread in the first case. Similarly the function `listen` passes control according to the state.

```
class Assert a where
  demand :: a -> Switch -> IORef (ValState a) -> a

instance Assert a => Assert [a] where
  demand [] s =  unsafePerformIO $ do
    copy s r []
    return []
  demand (x:xs) s = unsafePerformIO $ do
    r1 <- newIORef Untouched
    r2 <- newIORef Untouched
    copy s r (listen s r1 : listen s r2)
    return (demand x s r1 : demand xs s r2)
```

```
copy :: Switch -> IORef (ValState a) -> a -> IO ()
copy s r x =  do
  state <- readIORef r
  case state of
    Untouched -> writeIORef r (Evaluated x)
    DemandedByAssert -> do
      writeIORef r (Evaluated x)
      continueAssert s

listen :: Switch -> IORef (ValState a) -> a
listen s r = unsafePerformIO $ do
  state <- readIORef r
  case state of
    Untouched -> do
      writeIORef r DemandedByAssert
      continueMain s
      state <- readIORef r
      case state of
        Evaluated x -> return x
    Evaluated x -> return x
```

Finally we adapt the definition of the `assert` function to the concurrent setting. We use the `evalAssertion` of our first sequential implementation.

```
assert :: Assert a => String -> (a -> Bool) -> a -> a
assert n p x = unsafePerformIO $ do
  r <- newIORef Untouched
  s <- initSwitch
  forkIO (evalAssertion n p (listen s r) >> finishAssert s)
  continueAssert s
  return (demand x s r)
```

*Properties of the Implementation.* This implementation fulfils the central properties that evaluation of assertions does not influence the result of the main computation, no tested values are evaluated further than by the main computation, and a failed assertion is signalled before the main computation uses the faulty value. The implementation does not hold onto the data of all assertions until the end of the computation, because assertions are evaluated as early as possible without over-evaluation. The implementation does not need a wrapper function `runA`.

## 6 Stuck Assertions

We noted in Section 3 that lazy assertions must not be `False`. Computation of the underlying program should proceed not only if an assertion reduces to `True`, but also if computation of the assertion is *stuck*, that is the assertion cannot (yet) be reduced to a value at all. Consequently both the sequential and the concurrent implementation do not distinguish between assertions that reduce to `True` and assertions that are stuck.

Evaluation order can often be disregarded when considering the correctness of lazy functional programs. Lazy evaluation does, however, specify a mostly sequential semantics. The semantics of logical connectives such as `(&&)` are not symmetric. When the evaluation order demanded by an assertion does not agree with the evaluation order demanded by the underlying computation the assertion gets stuck.

### Example 1: revisited

Consider using our definition of `checkedWith` in the following expression:

```
1 ‘elem‘ (Union Empty 4 (Union Empty 2 Empty) ‘checkedWith‘ 6)
```

Both input set and result set of `checkedWith` are not strictly ordered, but no assertion fails! This is because only a part of each set is ever demanded by the computation so the assertion `strictlyOrdered` gets stuck. For example, of the input set only the part

```
Union _ 4 (Union Empty 2 _)
```

is demanded (where `_` indicates an undemanded expression). The computation of the function `strictlyOrdered` traverses the tree representation of the set in preorder. Hence it gets stuck on the unevaluated left subtree of the root `Union` constructor. Consequently it never makes the comparison `4 < 2` which would immediately make the assertion fail.

*Detecting the problem.* It would help to list at the end of all computation all assertions that are stuck. It is easy to extend our sequential implementation to do this. The concurrent implementation would need to be extended by a global list of blocked assertions, similar to the `finalisers` of the sequential implementation.

*A solution?* We could avoid sequentiality in the assertion by creating a separate concurrent thread for each atomic test. We can use `assert` to start new threads and call assertions within assertions.[2] In the following definition the sequential `(&&)`s have been replaced by `assert`s that do not actually check any property of their last arguments but start separate checking threads. This assertion is as eager as possible, because each `between` comparison is evaluated by a separate thread.

[2] We have to slightly modify the sequential implementation to ensure that assertions which are being added to the list `finalisers` during evaluation of assertions will still be eventually evaluated.

```
assertStrictlyOrdered :: Ord a => String -> Set a -> Set a
assertStrictlyOrdered n = assert n (soBetween Nothing Nothing)
  where
  soBetween _  _  Empty           = True
  soBetween lo hi (Union s1 x s2) =
    assert n (const (soBetween lo (Just x) s1)) $
    assert n (const (soBetween (Just x) hi s2)) $
    between lo hi x
  between lo hi x = maybe True (< x) lo && maybe True (> x) hi
```

Using assertions within assertions is a trick that should not be our final answer to the problem of stuck sequential assertions. An alternative implementation might use a new type that replaces `Bool` and provides a parallel logical conjunction.

## 7 Related Work

The work reported in this paper started as a BSc project. The second author's dissertation [5] includes an earlier version of concurrent assertions and discusses some example applications.

In Section 4 we adapted a technique first used in HOOD [4]. HOOD defines a class of types for which an `observe` function is defined. Programmers annotate expressions whose values they wish to observe by applying `observe` *label* to them, where *label* is a descriptive string. These applicative annotations act as identities with a useful side-effect: each value to which an annotated expression reduces — *so far as it is demanded by lazy evaluation* — is recorded, fragment by fragment as it is evaluated, under the appropriate label. The similarity of `observe` and `assert` is clear, but an important difference is that whereas `observe` records a sequence of labelled fragments for subsequent inspection or separate processing, `assert` reassembles them for further computation within the same Haskell program. A HOOD programmer can evaluate by inspection any assumptions or intentions they may have about recorded values, but this inspection is a laborious and error-prone alternative to machine evaluation of predicates.

Another well-established Haskell library for checking properties of functional programs is QuickCheck [1]. Properties are defined as boolean-valued functions, as in the example:

```
prop_ElemWith :: Set Int -> Int -> Bool
prop_ElemWith s x = x `elem` (s `with` x) == True
```

Evaluating `quickCheck prop_ElemWith` checks the property using a test suite of *pseudo-randomly generated* sets and elements as the values of `s` and `x`. The test-value generators are type-determined and they can be customised by programmers. QuickCheck reports statistics of successful tests and details of any failing case discovered. This sort of testing nicely complements assertions. QuickCheck properties are not limited to expressions that fit the context of a particular program point, and a separate testing process imposes no overhead when an application is run. But assertions have the advantage of testing values that actually occur in a program of interest, and provide a continuing safeguard against undetected errors.

Möller [6] offers a different perspective on the role of assertions in a functional language. The motivating context for his work is transformational program development; assertions carry parts of the specification and are subject to refinement. He assumes strict semantics, however, and does not consider the problem of assertions in a lazy language.

## 8 Conclusions and Future Work

Assertions, first used in call-by-value procedural languages, can be introduced in a way that fits with a call-by-need functional language. Assertions can be supported by a high-level library written in the functional language itself. The library can guarantee that assertions do not force evaluation beyond the needs of the underlying program, but programming assertions to fail as eagerly as possible despite this guarantee can be a delicate art.

There are still many areas to explore. We could define combinators to formulate assertions about functional and monadic values. We need experience in the use of assertions in larger applications. A failed assertion should output the evaluated part of its subject value. Using assertions in connection with Hat [9] would allow the causes of assertion failures to be traced — just as combined working with QuickCheck and Hat allows failed tests to be investigated [2]. We are looking for a more portable implementation that works with other Haskell systems than the Glasgow Haskell Compiler. We need to explore further the effect of assertions on the time and space performance of a program; in particular, the copying of values can cause a loss of sharing. Finally, we would like to garbage collect permanently stuck assertions.

**Acknowledgements**

Thanks to Dean Herington, Claus Reinke and Simon Peyton Jones for their contributions to a discussion on the Haskell mailing list about how to achieve data-driven concurrency.

## References

1. 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.
2. K. Claessen, C. Runciman, O. Chitil, J. Hughes, and M. Wallace. Testing and tracing lazy functional programs using QuickCheck and Hat. In *Lecture notes of the 4th Intl. Summer School in Advanced Functional Programming.* 40pp, to appear in Springer LNCS, 2002.

3. S. L. Peyton Jones (Ed.). Haskell 98: a non-strict, purely functional language. *Journal of Functional Programming*, 13(1):special issue, 2003.
4. A. Gill. Debugging Haskell by observing intermediate datastructures. *Electronic Notes in Theoretical Computer Science*, 41(1), 2001. (Proc. 2000 ACM SIGPLAN Haskell Workshop).
5. D. McNeill. Concurrent data-driven assertions in a lazy functional language. Technical report, BSc Project Dissertation, Department of Computer Science, University of York, 2003.
6. B. Möller. Applicative assertions. In J. L. A. van de Snepscheut, editor, *Mathematics of Program Construction*, pages 348–362. Springer LNCS 375, 1989.
7. S. L. Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions and foreign-language calls in haskell. In C. A. R. Hoare, M. Broy, and R. Steinbruggen, editors, *Engineering theories of software construction*, pages 47–96. IOS Press, 2001.
8. C. Runciman and N. Röjemo. Heap profiling for space efficiency. In J. Launchbury, E. Meijer, and T. Sheard, editors, *2nd Intl. School on Advanced Functional Programming*, pages 159–183, Olympia, WA, August 1996. Springer LNCS Vol. 1129.
9. M. Wallace, O. Chitil, T. Brehm, and C. Runciman. Multiple-view tracing for Haskell: a new Hat. In *ACM Workshop on Haskell*, 2001.

## Appendix: The class `Assert` and its Instances

In both the sequential and the concurrent implementation there is a class `Assert`. We need an instance of `Assert` for every type of value that we wish to make assertions about. To simplify the writing of new instances we define a family of $\texttt{demand}_n$ functions. For the concurrent implementation they are defined as follows:

```
demand0 :: Switch -> IORef (ValState a) -> a -> a
demand0 x s r = unsafePerformIO $ do
  copy s r x
  return x

demand1 :: (Assert b) => (b -> a) -> b
                      -> Switch -> IORef (ValState a) -> a
demand1 c x1 s r = unsafePerformIO $ do
  r1 <- newIORef Untouched
  copy s r (c (listen s r1))
  return (c (demand x1 s r1))

demand2 :: (Assert b, Assert c) => (c -> b -> a) -> c -> b
                                -> Switch -> IORef (ValState a) -> a
demand2 c x1 x2 s r = unsafePerformIO $ do
  r1 <- newIORef Untouched
  r2 <- newIORef Untouched
  copy s r (c (listen s r1) (listen s r2))
  return (c (demand x1 s r1) (demand x2 s r2))
```

Instances thus become short and easy to write:

```
instance Assert a => Assert [a] where
  demand [] = demand0 []
  demand (x:xs) = demand2 (:) x xs

instance (Assert a,Assert b) => Assert (a,b) where
  demand (x,y) = demand2 (,) x y

instance Assert Char where
  demand c = c `seq` demand0 c
```

The use of `seq` is needed in the last case where no pattern matching takes place to ensure that the value is always evaluated by the main thread, not the assertion thread.

Although this is an improvement, it will still be useful to use a tool such as DrIFT[3] to derive the often large number of instances needed in practice.

A different problem is that the class context of the function `assert` restricts its use in the definition of polymorphic functions. For Example 1 we obtain the type

```
checkedWith :: (Ord a, Assert a) => Set a -> a -> Set a
```

Users of HOOD seem to be able to live with a similar restriction.

For Hugs there is a special version of HOOD that provides a built-in polymorphic function `observe`. Likewise a built-in polymorphic function `assert` is feasible. Even better, since the implementations of `observe` and `assert` are based on the same technique, it is desirable to identify the functionality of a single built-in polymorphic function in terms of which both `observe`, `assert` and possibly further testing and debugging functions could be defined. A built-in polymorphic function removes both the annoying need for a large number of similar instances and the restricting class context.

[3] `http://repetae.net/john/computer/haskell/DrIFT/`