Lazy Assertions

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 disrupting 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 explore the subtle semantics of lazy assertions and describe sequential and concurrent variants of a method for checking lazy assertions. All variants are implemented in Haskell.

1 Introduction

A programmer writing a section of code often has in mind certain 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. Those beyond the expressive power of static types could perhaps be proved separately as theorems, but such a demanding approach is rarely taken. Instead of leaving key 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. If no assertion fails, the program runs just as it would normally, 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.

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.

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? 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 first outlines an implementation of lazy assertions that postpones their evaluation until the underlying program is finished; it then goes on to describe alternative implementations in which each assertion is evaluated by a concurrent thread. Section 5 uncovers a residual problem of sequential demand within assertions. Section 6 gives a brief account of our early experience using lazy assertions in application programs. Section 7 discusses related work. Section 8 concludes and suggests future work.

2 Eager Assertions Must be True

A library provided with the Glasgow Haskell compiler[1] already includes a function `assert :: Bool -> a -> a`. It is so defined 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.

Example: sets 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 $x \in s$.

[1] `http://www.haskell.org/ghc`

```
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, which is what 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 does matter. If assertion-checking forces evaluation

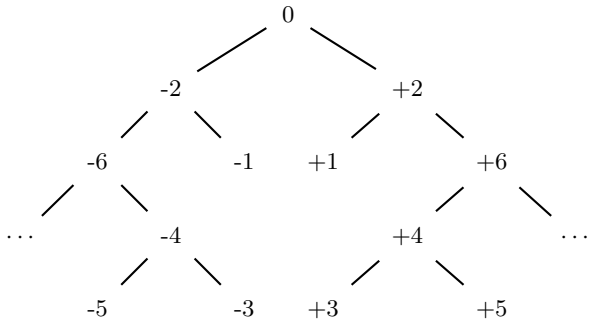

Fig. 1. A tree representation of the infinite set of integers. Each integer i occurs at a depth no greater than $2\log_2(\text{abs}(i) + 1)$. Differences between adjacent elements on leftmost and rightmost paths are successive powers of two.

it could 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!

Even where eager assertions terminate they may consume time or space out of proportion with normal computation. Also, assertions are often checked in the hope of shedding light on a program failure; it could be distracting to report a failed assertion about values that are irrelevant as they were never needed by the failing program.

3 Lazy Assertions Must Not Be False

So 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.

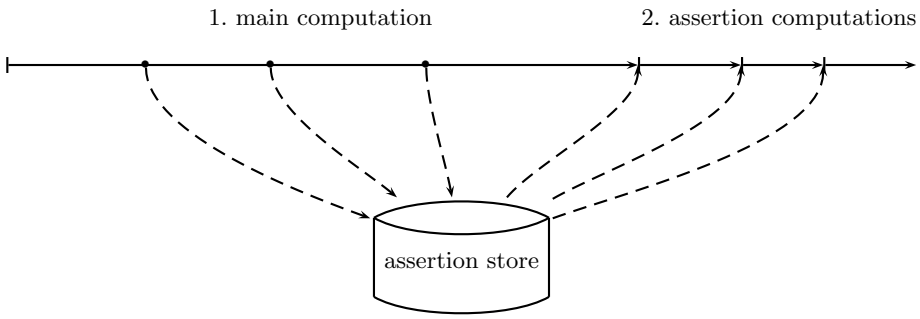

Fig. 2. Delayed Assertions in Time

Example revisited

If we had an implementation of this lazy `assert`, how would it alter the ordered-tree example? 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 a `checkedWith` application proceeds more like a normal application of `with`. Even 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.

4 Implementation

Having established the benefits of lazy assertions we now turn to the question of how they can be implemented in Haskell. We develop an assertion library in steps: we start with a simple version, criticise it, and then refine it to the next version.

4.1 Delayed Assertions

We have to ensure that the evaluation of the assertions cannot disturb the evaluation of the underlying program. A very simple idea for achieving this is to evaluate all assertions *after* termination of the main computation.

Figure 2 illustrates the idea. The main computation only evaluates the underlying program and collects all assertions in a global store. After termination of the main computation assertions are taken from the store and evaluated one after the other.

We are certain that lazy assertions cannot be implemented within pure Haskell 98. In particular we need the function `unsafePerformIO :: IO a -> a` to perform actions of the IO monad without giving `assert` a monadic type. We aim to minimise the use of language extensions and restrict ourselves to extensions supported by most Haskell systems. Our implementation is far more concise and potentially portable than any modification of a compiler or run-time system could be.

Which extensions do we need for delayed assertions? Extended exceptions enable a program to catch all erroneous behaviour of a subcomputation. They ensure that all assertions are evaluated, even if the main computation or any other assertion evaluated earlier fails. A mutable variable of type `IORef` implements the global assertion store. Finally `unsafePerformIO :: IO a -> a` enables us to implement `assert` using exceptions and mutable variables [7].

Properties of the Implementation. This simple implementation does not prevent an assertion from evaluating a test argument further than the main computation did. 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 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 assertion uses the result of the function `listen`. 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, then 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 `v` (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 `Assert` and the type of `assert` becomes `Assert a => String -> (a -> Bool) -> a -> a`. Appendix A gives the details of the implementation.

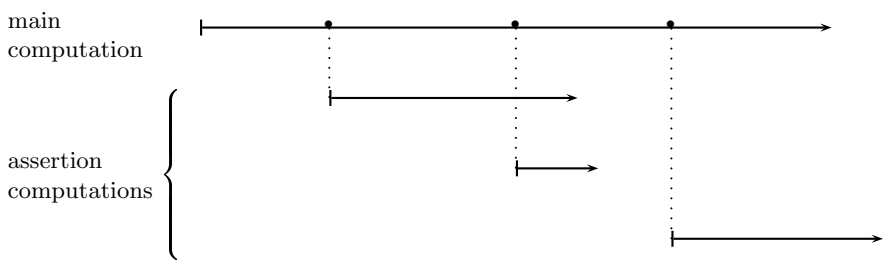

Fig. 3. Concurrent Assertions in Time

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!

4.3 Concurrent Assertions

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. We require a further extension of Haskell 98: Concurrent Haskell [7].

Figure 3 illustrates the idea. Each evaluation of `assert` in the main computation starts a new thread for evaluating the assertion itself. As before, the value tested by an assertion is copied as it is demanded by the main computation and the copy is used by the assertion. Replacing the `IOVar` shared by a `demand/listen` pair by an `MVar` synchronises the assertion thread with the demand of the main computation. The assertion thread has to wait when it tries to evaluate parts of the copy that do not (yet) exist.

Properties of the Implementation. Concurrency ensures that even if the main computation runs into an infinite loop, a failed assertion will be reported. In general failed assertions may be reported earlier. However, there is no guarantee, because the scheduler is free to evaluate assertions at any time. They may — and in practice often are — evaluated after the main computation has terminated.

4.4 Priority of Assertions

To solve the problem we need to give assertion threads priority over the main computation. Unfortunately Concurrent Haskell does not provide threads with

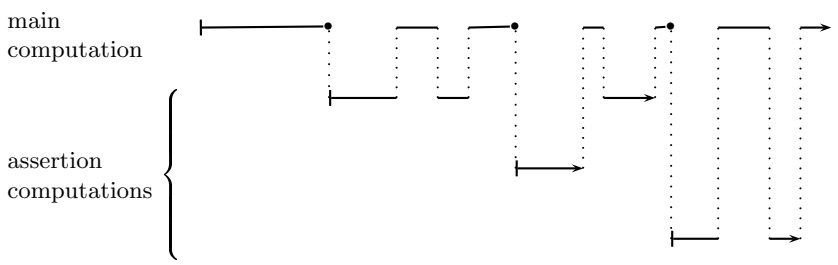

Fig. 4. Concurrent Assertions with Priority in Time

different priorities. However, coroutining enables us to give priority to assertions. We explicitly pass control between each assertion thread and the main thread. 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. Figure 4 illustrates the idea and Appendix B gives the details of the implementation which uses semaphores to pass control.

Properties of the Implementation. Coroutining ensures that a failed assertion is reported before the main computation uses the faulty value. Furthermore, the implementation does not hold onto all data needed by assertions until the end of the computation, because assertions are evaluated as early as possible without over-evaluation. However, assertions that cannot be fully evaluated are still live until the end of the whole computation.

4.5 Garbage Collecting Stuck Assertions

When a tested value is no longer reachable from the main computation thread, it will no longer be demanded by the main computation and hence the assertion thread is permanently stuck. We extend the coroutining implementation with finalisers [8] that kill an assertion thread when its value is no longer reachable from the main computation thread.

Properties of the Implementation. This implementation reduces the requirements for space and threads.

4.6 Conclusions

During development we identified the following important properties of a lazy assertion library.

– Evaluation of assertions does not influence the main computation.
– Assertions do not evaluate values further than the main computation does.
– A failed assertion is reported before the main computation uses the faulty value.
– The requirements for space and threads are minimised.

For each property we developed a new implementation. Unfortunately we find that the implementations using coroutining violate the first property. Suppose we define `assertFun` as follows to assert a relation between the argument and result of a function.

```
assertFun :: (Assert a, Assert b)
          => String -> (a->b->Bool) -> (a->b) -> (a->b)
assertFun n p f i = o'
  where
  (i',o') = assert n (uncurry p) (i,f i')
```

This cyclic definition works fine with all but the coroutining implementations of `assert`. With coroutining a deadlock occurs because the assertion thread waits for the input `i'` of the function which has to be produced by the assertion thread itself.

We conclude that the concurrent implementation without priorities is the most useful implementation we have. We have to aim for a concurrent implementation with priorities and garbage collection of stuck assertions that controls threads less restrictively than coroutining.

5 Sequential Semantics causes 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 our implementations 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 order in which an assertion demands components of a data structure does not agree with the order in which the main computation demands the components of that data structure, the assertion can get stuck.

Example revisited again

Consider evaluation of the following expression:

```
5 `elem` (assert "ordered" strictlyOrdered
  (Union (Union Empty 2 Empty) 3 (Union Empty 1 Empty)))
```

The given set is not strictly ordered, but no assertion fails! This is because only the part

```
Union _ 3 (Union _ 1 Empty)
```

of the set is ever demanded by the computation (_ 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 `3 < 1` 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 implementations to do this.

A solution? We could avoid sequentiality in the assertion by creating a separate assertion for each atomic test. 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 assertions. This assertion is as eager as possible, because each `between` comparison is separate.

```
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
```

These assertions within assertions work with all implementations except for the coroutining ones. Again coroutining leads to a deadlock.

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.

6 Larger Applications

As yet we have only tried out lazy assertions in a few programs of modest size. We note briefly some of our experience with two of these programs.

Clausify

The `clausify` program puts propositions represented using the type

```
data Prop = Sym Char      | Neg Prop
          | Dis Prop Prop | Con Prop Prop
          | Imp Prop Prop | Eqv Prop Prop
```

into clausal form, by a composition of several stages. We found it convenient to write assertions using an auxiliary function

```
propHas :: (Prop -> Bool) -> Prop -> Bool
```

defined so that `propHas t p` applies test `t` both to `p` itself and to all `Prop`s that `p` contains. We also find a use for implication lifted to predicate level

```
implies :: (a -> Bool) -> (a -> Bool) -> (a -> Bool).
```

After successive stages, the following assertions should hold, cumulatively:

1. `propHas (\p -> not (isImp p || isEqv p))`
   `Imp` and `Eqv` are eliminated.
2. `propHas (isNeg ‘implies‘ (\(Neg q) -> isSym q))`
   In addition, `Neg (Sym _)` is the only permitted form of negation.
3. `propHas (isDis ‘implies‘ \(Dis p q) -> not (isCon p || isCon q))`
   Further, no `Con` occurs within a `Dis`.

If a fault is introduced into any of these stages, so that it fails to normalise a proposition as it should, the result is typically a pattern-matching failure in a later stage. We found that lazy assertion checking often reports the failure in the earlier stage, but sometimes inconclusively reports the relevant assertion as stuck. To minimise stuckness one has to think carefully about evaluation order in assertions.

Pasta

Further issues arose when we introduced lazy assertions in `pasta`, an interpreter for a small imperative language with dynamic data structures. Our goal was to assert a data invariant for a moderately complex data structure representing the environment and store:

```
data EnvStore = ES {sig   :: Signature,
                    ops   :: [Operation],
                    scope :: [Name],
                    stack :: [Value],
                    heap  :: [StructVal]}
```

To make assertions over `EnvStore` values would seem to require an `Assert` instance for `EnvStore`. But because of the various component types (and *their* component types etc.) this would mean a fair bit of work in several different modules. As the invariant properties relate only the scope, stack and heap, we avoid much of this work by embedding the invariant assertion in a smart constructor like this:

```
es si o sc st h = ES si o sc' st' h'
  where
  (sc',st',h') = assert "ES invariant" dataInv (sc,st,h)
```

The details of `dataInv` are not important here. The most surprising result was that *none* of the `dataInv` assertions was ever fully evaluated! The explanation is that the interpreter uses `EnvStore` values in a single-threaded way, and each state change only involves accessing a small part of the relevant `EnvStore`. Since lazy assertions only check the parts actually used by the program, they never get to check a complete `EnvStore` structure. The contrast with an eager data invariant is striking.

7 Related Work

The work reported in this paper started as a BSc project. The second author's dissertation [5] describes experiments with an earlier version of concurrent assertions.

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.

HOOD does not require threads or non-trivial delayed computations. A fragment of a value is recorded just when it is demanded. It would be nice if the implementation of assertions could be that simple. However, an assertion usually relates several fragments of a value, for example, it may compare two numbers in a tree. The assertion can only be checked when the last of the two numbers becomes available, no matter in which order they are demanded by the main computation. Additionally, the demands of the assertion predicate can only be determined by applying it to an argument.

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 also be useful in a call-by-need functional language; but they should be constrained appropriately. The key requirement is that assertion-checking never forces evaluation beyond the needs of the underlying program.

We have shown how appropriately lazy assertions can be supported by a high-level library. Our account has been based on experimental prototypes developed using the Glasgow Haskell Compiler, and these prototypes do rely on some of the language extensions this compiler supports. We would prefer to have a more portable library.

It would be easy to extend the reports from a failed assertion to include the evaluated part of its subject value. To allow the causes of assertion failures to be traced, we may eventually support the use of assertions in connection with Hat [9, 2].

We do need more experience with the use of lazy assertions in larger applications. So far we have found that expressing assertions in the functional language itself is a pleasant task, but it might be useful to include a few standard combinators in the library, especially for making assertions about functional (and perhaps monadic) values. Programming lazy assertions to fail as eagerly as possible can be tricky, and it is not yet clear whether suitable abstractions such as concurrent logical operators will help. We also need to explore further the effect of assertions on the time and space performance of a program, particularly as the copying of values can cause a loss of sharing. Pragmatics are not easily hidden by abstraction!

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. Simon L. Peyton Jones, Simon Marlow, and Conal Elliott. Stretching the storage manager: Weak pointers and stable names in haskell. In *Implementation of Functional Languages, 11th International Workshop, IFL'99*, volume 1868 of *LNCS 1868*, pages 37–58, 2000.
9. M. Wallace, O. Chitil, T. Brehm, and C. Runciman. Multiple-view tracing for Haskell: a new Hat. In *ACM Workshop on Haskell*, 2001.

A Sequential Implementation: Delayed Assertions Avoiding Over-Evaluation

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.

```
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)
```

Only evaluation of `evalAssertion n p x` actually evaluates the assertion of name `n` and predicate `p` with test argument `x`. The function `evalAssertion` has to catch exceptions to ensure that an exception in one assertion does not prevent the remaining pending assertions from being tested. The function `evalAssertion` also 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))
```

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.[2]

```
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
```

Finally the functions `demand` and `listen` implement the demand driven copying of a tested value by the main computation for the assertion.

```
data ValState a = Blocked | Unblocked a

class Assert a where
```

[2] 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`.

```
  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
```

B Concurrent Implementation: Assertions with Priority

To control the running status of a pair of threads we introduce a `Switch` of two binary semaphores and associated functions for passing control. The function `waitQSem` blocks a thread until a 'unit' of a semaphore becomes available, and `signalQSem` makes a 'unit' available.

```
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 definitions of the function `assert` and `evalAssertion` to the concurrent setting. The function `forkIO` starts a new thread.

```
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)

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 ++
              " failed with exception: " ++ show e)
```

This implementation does not need a wrapper function `runA`.

C The class `Assert` and its Instances

In both sequential and concurrent implementations 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, a tool such as DrIFT [3] is still useful to derive what may be a large number of instances.

A different problem is that the class context of the function `assert` restricts its use in the definition of polymorphic functions. For our running example 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/`