A Pattern Logic for Prompt Lazy Assertions in Haskell*

Olaf Chitil
University of Kent, UK
oc@kent.ac.uk

Frank Huch
CAU Kiel, Germany
fhu@informatik.uni-kiel.de

Abstract. Assertions test expected properties of run-time values without disrupting the normal computation of a program. Here we present a library for enriching programs in the lazy language Haskell with assertions. Expected properties are written in an expressive *pattern logic* that combines pattern matching with logical operations and predicates. The presented assertions are lazy: they do not force evaluation but only examine what is evaluated by other parts of the program. They are also prompt: assertion failure is reported as early as possible, before a faulty value is used by the main computation.

1 Introduction

Large programs are composed of algorithms and numerous (more or less) abstract data types which interact in complex ways. A bug in the implementation of a basic data structure can result in the whole program going wrong. Such a bug can be hard to locate, because the faulty data structure may not be part of the wrong result, it may just be an intermediate data structure. Even worse, the program may produce wrong results for a long time before the user even notices.

Testing abstract data types exhaustively is difficult. However, interesting test cases often occur when data structures are used within other algorithms. Hence it is a good idea to check for bugs in basic data structures and functions during the execution of larger programs. Using *assertions* is a common approach to do so. The programmer specifies *intended properties* of data structures and functions by writing assertions. During program execution, these assertions are tested and failure of an assertion is reported to the programmer. Examples of assertions are restricting the square root function to positive arguments or the property of being sorted for a search tree.

The Glasgow Haskell Compiler (GHC) already provides the possibility to define assertions:

```
assert :: Bool -> a -> a
```

The first argument is the asserted property. If this property evaluates to True, then `assert` behaves like the identity function. Otherwise, an error is reported with detailed information about the source code position of the failed assertion. For example, consider an assertion that checks whether a list is sorted:

* 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.

```
checkSorted :: Ord a => [a] -> [a]
checkSorted xs = assert (sorted xs) xs

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

Unfortunately `assert` is strict in its Boolean argument which clashes with Haskell's laziness. The asserted property is evaluated and the tested data structure is evaluated as far as necessary to decide the property. Hence, programming with assertions will result in strict programs with loss of the expressive power of laziness, e.g., the use of infinite data structures.

We conclude that assertions in lazy languages should respect laziness. They should only be evaluated as far as possible, i.e., an assertion should only be checked for the part of the data structure which is evaluated during the computation. A first approach for lazy assertions is [2]. It is based on

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

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` ensures that only the context in which the application of `assert` appears determines how far the tested data structure is evaluated. Only the evaluated part is passed as argument to the predicate.

We can redefine `checkSorted` as follows:

```
checkSorted xs = assert "sorted" sorted xs
```

Applying `checkSorted` to the list `[1,3,2,4]` yields:

```
Assertion (sorted) failed: 1:3:2:_
```

The failure is reported as early as possible, before the whole list is evaluated. However, the approach of [2] has a major drawback. If we evaluate only the tail of the observed list, no failure occurs, although the evaluated part of the observed data structure is not sorted:

```
> tail (checkSorted [1,3,2,4])
[3,2,4]
```

The reason for this behaviour is that the function `(&&)` used in the definition of the predicate `sorted` is sequentially defined. The assertion is suspended on checking the sorted property for the first two elements of the list. The conjunction is never evaluated to `False`, although there are two elements in the evaluated part which are not in order.

In practice, many lazy assertions are suspended exactly for this reason. Many asserted properties may not hold for evaluated parts of data structures, but the assertions do not fail and hence, the programmer wrongly believes their program to be correct. The evaluation of an assertions involves a *sequential* evaluation order, which may not be related to the evaluation order of the program generating/evaluating the data structures.

In this paper we introduce a new approach for lazy assertions. The basic idea is to define assertions by means of a pattern logic instead of arbitrary Haskell functions. In this logic, we express properties with parallel versions of `(&&)` and `(||)`. If any of the arguments of such a parallel operator makes the whole assertion fail, then this is reported independently of the other parts of the assertion. Furthermore, our assertions are checked as early as possible, which we call *promptness*. Whenever a new part of a data structure is required by the main computation, assertions are checked for this part and any assertion failure is reported before a faulty value is used by the main computation.

Although in some cases this approach may be more complicated than defining assertions within the programming language Haskell itself, there is also an opportunity. Our pattern logic is more a specification language than a programming language. Hence, properties are asserted in a style that is completely different to ordinary programs. So it is unlikely that programmers will make the same mistakes in the assertions as in the program, which may happen easily using the same language for programming as for specifying properties.

Beside reporting failed assertions, reporting how many and which assertions have succeeded may also be useful. We collect succeeded assertions in a file, so that the programmer can later analyse which assertions succeeded. However, not every assertion is supposed to succeed in the presence of laziness. The user must be aware that in many cases checking assertions suspends and cannot be decided on the evaluated parts of the data structures. This behaviour is even required when a property shall be tested on a never fully evaluated infinite data structure. However, if an assertion fails because of any part of an evaluated data structure, then this is reported immediately.

Our assertions have the following properties:

- They do not modify the lazy behaviour of a program.
- Whenever some part of a data structure is evaluated and this part violates an asserted property, this is promptly reported to the programmer.
- 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.

In Sections 2 to 5 we explain how to use our pattern logic by means of examples. Section 6 outlines how the implementation works. In Section 7 we discuss related work and we close in Section 8.

2 Patterns and Quantification

In the following sections we introduce our pattern logic step by step and justify our design decisions through examples.

2.1 Patterns

Pattern matching is a powerful feature of modern functional languages. The pattern is a kind of prototype of a function's argument. For example, it allows a simple definition of a function that tests whether a list has exactly two elements:

```
hasTwoElements :: [Int] -> Bool
hasTwoElements (_:_:[]) = True
hasTwoElements _        = False
```

We can define a function that is basically the identity function on lists but additionally asserts that the argument has exactly two elements as follows:

```
twoElements :: [Int] -> [Int]
twoElements = assert "two elements" (p_ <:> p_ <:> pNil)
```

So what are the new functions used in this definition? We cannot use built-in pattern matching for prompt lazy assertions and we do not want to extend the language Haskell. Therefore, we implement our pattern logic using an abstract type constructor `Pat`. We provide functions for constructing `Pat`s:

`p_ :: Pat a` is the wildcard pattern that matches everything;
`pNil :: Pat [a]` and `(<:>) :: Pat a -> Pat [a] -> Pat [a]`

construct patterns that match the two data constructors of the list type. Using these pattern constructors we can write `p_ <:> p_ <:> pNil` to express a `Pat` similar to the pattern `_:_:[]` used in the definition of `hasTwoElements`. For every predefined data type appropriate patterns are defined, e.g., `pNothing` and `pJust` for matching `Maybe` values and `pPair` for matching pairs.

The assertion itself is expressed with

```
assert :: Observe a => String -> Pat a -> a -> a
```

The type of any value we make assertions about has to be an instance of class `Observe`, whose rôle is explained later.

Whereas `hasTwoElements` forces the evaluation of the list constructors of its argument to perform pattern matching, `twoElements` is lazy: the argument is only evaluated as far as its result is demanded by the caller of `twoElements`.

In many cases it will be useful to combine patterns by means of the logical conjunction and disjunction operators:

```
(|||) :: Pat a -> Pat a -> Pat a      (&&&) :: Pat a -> Pat a -> Pat a
```

For instance, we can now define an assertion which expresses that a list contains less than two elements:

```
shortList :: [a] -> [a]
shortList = assert "length less than two" (pNil ||| p_ <:> pNil)
```

2.2 Context Patterns

When specifying properties of large data structures, it is not sufficient to match a finite initial part of the data structure. We would like to be able to match patterns in arbitrarily deep contexts, for example, to select an arbitrary element of a list. Hence we provide *context patterns* within our pattern logic. The pattern constructor

```
pListC :: Pat [a] -> Pat [a]
```

matches its argument pattern against arbitrary sublists of a list. For example

```
oneTrue :: [Bool] -> [Bool]
oneTrue = assert "True in list" (pListC (pTrue <:> p_))
```

asserts that there exists an element `True` in the argument list.

2.3 Universal and Existential Quantification

Why does the preceding example assert that there *exists* an element `True`? Could it not mean that *all* elements should be `True`? Indeed we will sometimes want to assert a property for all sublists and sometimes want to assert that there exists a sublist with a given property. Hence we introduce the quantifier patterns

```
forAll, exists :: Pat a -> Pat a
```

which change the meaning of context patterns within their scope. So

```
exists (pListC (pTrue <:> p_))
```

asserts that there exists an element `True` whereas

```
forAll (pListC ((exists pTrue) <:> p_))
```

asserts that all list elements are `True`.

Why is there a nested `exists` in the last example? Because quantifiers do not only change the semantics of context patterns, but also of normal patterns. Within the scope of `forAll` a constructor pattern such as `pTrue` matches any other constructor. Because of the quantifier `forAll` the context pattern `pListC` has to match *all* sublists with its argument pattern. In any finite list one sublist will be `[]`. We could list this alternative in our definition:

```
forAll (pListC (pTrue <:> p_ ||| pNil))
```

This is acceptable for lists, but not for more complex types with more constructors, such as abstract syntax trees. We would have to add a disjunction for every constructor and the size of assertions would blow-up unacceptably. Therefore we decided that within the scope of `forAll` a pattern built from `(<:>)` also matches the empty list. In contrast, in an existential context the pattern describes which structure is supposed to exist. Hence, non-matching sub-data-structures should not match the pattern inside `exists`. So within the scope of `forAll` the pattern `exists pTrue <:> p_` matches both the empty list and a non-empty list that does start with `True`. In contrast, the pattern `pTrue <:> p_` also matches a non-empty list starting with `False`. Additionally, the dependence of the pattern semantics on quantification becomes crucial in the context of predicates with several arguments, as we will show in Section 3.4.

The function `assert` implicitly surrounds its pattern by `exists`. Hence in the preceding subsection the pattern context is existentially quantified.

3 Predicates

Pattern matching cannot express properties of primitive types, such as a number being positive or a number being greater than another. For expressing such properties, Haskell enriches standard pattern matching with guards, in which the programmer specifies restrictions for the bound variables.

3.1 Unary Predicates

Because we cannot define a new variable binding construct within Haskell, we cannot bind normal variables in our patterns. Instead, we introduce a new pattern `val` that represents binding a variable to a value. To check a property of such a "variable" we provide a function `check`.

For example, we define an assertion that checks whether a number is positive:

```
posInt :: Int -> Int
posInt = assert "positive" (check val (>0))
```

Similarly, we can define a more complex assertion that checks whether all elements of a list are positive:

```
allPos :: [Int] -> [Int]
allPos =
  assert "all positive" (forAll (pListC ((check val (>0)) <:> p_)))
```

3.2 Predicates with several Arguments

Unary predicates are not very expressive. For instance, it is not possible to compare two elements of a data structure, as it is necessary for expressing the property of being sorted. Hence we extend the function `check` so that values from different `val`s can be compared in a predicate:

```
sortedList :: [Int] -> [Int]
sortedList = assert "sorted"
    (forAll (check (pListC (val <:> (pListC (val <:> p_)))) (<=)))
```

We select two elements within a list (respecting their positions in the list) by means of two list contexts, and check whether these two elements are in order. The assertion is checked for every possible combination of elements in the list. Evaluating `sortedList [2,4,6,3,5]`, the following failure is reported:

```
Assertion (sorted) failed: 2:4:6:3:_
```

The result of the application is the list itself. For printing this list, the list has to be evaluated from left to right. When the list element 3 is evaluated, the assertion fails. The list elements which cause the assertion to fail are highlighted. Because the remaining list is not evaluated at all, an underscore is presented to the user for the unevaluated tail of the list. With a different evaluation order of the values within the list other failure positions may be reported. However, our assertions are prompt. When an assertion fails during the evaluation of a data structure, this is directly reported to the user. The data structure is not evaluated any further.

Checking `sortedList` is expensive in time ($\mathcal{O}(n^2)$, where n is the length of the list). Using the transitivity of `(<=)`, we can define a linear variant instead:

```
sortedLin :: [Int] -> [Int]
sortedLin = assert "sortedLin"
    (forAll (check (pListC (val <:> val <:> p_)) (<=)))
```

However, assertions should be seen as high-level specifications for which it is more important to be understandable and correct than to be efficient. Furthermore, this more efficient implementation has another drawback. If only every second element of the list is evaluated, then `sortedLin` will not compare any list element, i.e., for a list which is only evaluated to `1:_:2:_:1:_` `sortedLin` does not fail, whereas the less efficient assertion `sortedList` would fail. On the other hand, in practice evaluation orders like this one are uncommon and failure of `sortedLin` will in most cases be detected as early as failure of `sorted`.

3.3 The Pattern Type

When introducing predicates with more than one argument, we have to extend the definition of patterns (`Pat`) as well. Applying `check` to a pattern and a predicate function, we have to guarantee that the predicate takes as many arguments as `val`s occur in the pattern. Furthermore, the type of each value matched by `val` and the corresponding argument of the predicate must agree. In other words, `check` should have a type like

```
check :: Pat a (b1,...,bn) -> (b1->...->bn->Bool) -> Pat a ()
```

where $b_1, \ldots, b_n$ are the types of the values matched by `val`s. How can such a type be expressed within Haskell 98? We want `check` to work with predicates of any arity. Even a set of `check` functions indexed by arity would not do as a first take at the type of a simple constructor pattern demonstrates:

```
(<:>) :: Pat a (b1,...,bn) -> Pat [a] (bn+1,...,bm)
         -> Pat [a] (b1,...,bm)
```

How shall we handle all these varying numbers of arguments collected by `val` for the predicate tested by `check`? The solution is to extend the type constructor `Pat` not by one but by two type arguments. The first is the type of a predicate passed as input to the pattern and the second is the type of a predicate resulting from the pattern. We revise the types as follows:

```
check :: Pat a (b1->...->bn->Bool) Bool -> (b1->...->bn->Bool)
         -> Pat a Bool Bool

(<:>) :: Pat a (b1->...->bm->Bool) (bn+1->...->bm->Bool) ->
         Pat [a] (bn+1->...->bm->Bool) Bool
         -> Pat [a] (b1->...->bm->Bool) Bool
```

These are still not Haskell 98 types, but they are instances of types that we can use:

```
check :: Pat a b Bool -> b -> Pat a c c
(<:>) :: Pat a b c -> Pat [a] c d -> Pat [a] b d
```

So the second type argument of `Pat` is the type of a value passed into the pattern and the third type argument is the type of a value passed back out of the pattern, if the pattern matches. We always use patterns for which these passed values are predicates or simply Boolean values.

The type of `check` expresses that the predicate of type `b` has to be applied to all its arguments in the pattern to return a Boolean value. The variable bindings within `check` are encapsulated. Also, while `check` tests the predicate for its argument pattern, it also accepts a predicate as input which it passes back unchanged, if the pattern matches.

We have the following type for the variable pattern:

```
val :: Pat a (a -> b) b
```

This type expresses that the input function is applied to the matched value and the result is passed back. *We do not discuss all modified type signatures here.*

To make our assertions lazy, `val` can only be performed if the selected data structure is fully evaluated. Otherwise the predicate would be tested on partially evaluated values, which could involve further evaluation destroying the laziness of our assertions. However, the pattern `val` is usually used for values of primitive types, which cannot be evaluated partially at all.

3.4 Example: Equal Sets

Let us define the property that two sets (implemented as unordered lists without repeated items) contain the same elements. A simple way to describe this property would be the following:

> For each element of the first list, there exists an equal element in the second list and
> for each element of the second list, there exists an equal element in the first list.

Using our quantifiers, the first of these two assertions can easily be defined as follows:

```
subset :: ([Int],[Int]) -> ([Int],[Int])
subset = assert "already subset"
    (check (pPair (forAll (pListC (val <:> p_)))
                  (exists (pListC (val <:> p_))))
           (==))
```

The quantifiers are nested with respect to the order in which they appear within the linearly written formula. Hence, for every element of the first list an equal element within the second list has to exist. Expressing the other direction is more difficult, because the nesting of quantifiers (`forAll exists`) has to be applied in the reverse order of the tuple elements. We need to first select any element of the second list and then check whether there exists the same element within the first list. This can be expressed by matching the same data structure twice, by means of a modified conjunction operator

```
(+++) :: Pat a b c -> Pat a c d -> Pat a b d
```

which applies both argument patterns to the same data structure and collects all `val`s within the two argument patterns (all combinations — like a product) to apply a predicate to these by means of `check`. Using this operator, we can define the complete assertion as:

```
equalSets :: ([Int],[Int]) -> ([Int],[Int])
equalSets = assert "equal sets"
   (check (    pPair (forAll (pListC (val <:> p_)))
                     (exists (pListC (val <:> p_)))
          &&& (    pPair p_ (forAll (pListC (val <:> p_)))
               +++ pPair (exists (pListC (val <:> p_))) p_)))
              (==))
```

Evaluation of `equalSets ([1,2,3],[3,2,2,1])` just yields the tuple of sets, whereas the call `equalSets ([1,2,3],[3,2,4,2,1])` aborts with the message:

```
Assertion (equal sets) failed: (1:(2:(3: [] )),3:(2:( 4 :_)))
```

For the element 4 of the second list, no element was found in the first list. In the presence of existential properties it is not so easy to show the programmer where an assertion failed. The first list does not contain the element 4. Marking the first list completely would present the reason for the failure of the existentially quantified part. However, this would often mean that the whole data structure is marked. Hence, we decided to mark only that part of the data structure,

at which the failure of the existential pattern is observed. To distinguish these sub-terms from those causing failure of a universally quantified `val` we use a lighter colour for marking. In this application the lists were evaluated from left to right. As a consequence, the empty list made the decision that the assertion fails possible and we mark it. If the elements of the list were evaluated in another order, another element might be highlighted.

This example also shows why the design decision of making the constructor pattern semantics dependent on quantification is crucial. If the constructor pattern `<:>` does not match the empty list within a `forAll` context, then we have to add patterns for all other constructors (the empty list), i.e., replace the pattern

```
(forAll (pListC (val <:> p_)))
```

by the disjunction

```
(forAll (pListC ((val <:> p_) ||| pNil)))
```

Unfortunately, this is not possible and results in a type error. The pattern `pNil` does not yield a value for which we can check whether it occurs in the other list.

4 Further Patterns and Assertion Features

4.1 Functions

So far, our approach allows programmers to annotate arbitrary data structures with assertions. However, where should a programmer add such assertions? To express pre- and post-conditions, it would be nice to add assertions directly to functions. Furthermore, in a higher-order language, it should be possible to add assertions to functional arguments, functional return values, and functions within data structures as well.

In our pattern logic we handle functions just like any other data structure. The idea is that a function can be seen as a set of argument/result pairs which are matched by the function pattern

```
(-->) :: Pat a c d -> Pat b d e -> Pat (a -> b) c e
```

The first argument of `(-->)` is matched against the argument the function is applied to. The second argument is matched against the function result. An assertion for functions will usually contain predicates relating arguments and results. Hence, its type is similar to any pattern constructor of arity two.

Because again `b` can be a functional type, patterns for functions with higher arity can be defined by nested `(-->)` applications. As an example we consider the greatest common divisor (*gcd*) of two numbers. A reasonable assertion for gcd is that the result is a factor of both arguments:

```
gcd :: Int -> Int -> Int
gcd = assert "result is factor of arguments"
  (forAll (check (val --> val --> val)
                 (\x y res -> mod x res==0 && mod y res==0))) gcd'

gcd' :: Int -> Int -> Int
gcd' n m = let r = n `mod` m in if r == 0 then m else gcd n r
```

The algorithm is implemented by the function `gcd'`. For the assertion, we add a wrapper `gcd` which checks every application of `gcd'`. The function works correctly for many arguments, but we finally get a report like:

```
Assertion (result is factor of arguments) failed: 6 -> 9 -> 6
```

The function `gcd` applied to the arguments 6 and 9 yields 6, which is wrong, because 6 is not a factor of 9. The reason is the wrong argument of `gcd` in the recursive call to `gcd`: we wrote `n` instead of `m`. After fixing the bug, the assertion is always satisfied.

In contrast to data structures, which are only evaluated once during the computation, functions can be applied many times. The assertion is checked for each application and any failure is reported to the programmer.

The definition of `gcd` demonstrates how programmers should add assertions to their functions. The defined function is renamed (here to `gcd'`) and a wrapper with the original name (`gcd`) is defined.

Because `(-->)` is just a standard pattern constructor, its usage is not restricted to top-level function definitions. We can also use it for asserting properties of functional arguments and results as well as for functions occurring within data structures.

4.2 Negation and Implication

Finally we add negation to the logic: `neg :: Pat a b Bool -> Pat a b Bool`

We restrict negation to Boolean formulas, because using values selected by both negated and non-negated patterns in the same predicate does not make sense. We can, for example, define implication in the common way:

```
(==>) :: Pat a b Bool -> Pat a b Bool -> Pat a b Bool
(==>) pat1 pat2 = neg pat1 ||| pat2
```

4.3 Positions in Data Structures

For tree-like data structures it can be useful to compare positions of selected values in the structure. We provide positional information by means of

```
valPos :: Pat a ((Pos,a) -> b) b
```

where Pos is an abstract data type which can be compared by functions such as

```
moreLeft :: Pos -> Pos -> Bool          above :: Pos -> Pos -> Bool
```

p1 `` `moreLeft` `` *p2* is true iff in an in-order traversal of the data structure *p1* is reached before *p2* is reached. *p1* `` `above` `` *p2* is true iff position *p2* is within the substructure at position *p1*. For example, using positions, the property of being sorted can be defined as follows:

```
sortedPos = assert "sortedPos"
   (forAll (check (pListC (valPos <:> p_) +++ pListC (valPos <:> p_))
                  (\ (p1,x1) (p2,x2) -> p2 `moreLeft` p1 || x1<=x2))
```

We non-deterministically select two elements of the list and compare them taking their positions into account.

```
class Observe a where
  observe :: a -> Obs a

o0 :: a -> String -> Obs a
o1 :: Observe a => (a -> b) -> String -> a -> Obs b
o2 :: (Observe a,Observe b) => (a -> b -> c) ->
      String -> a -> b -> Obs c
o3 :: (Observe a,Observe b,Observe c) => (a -> b -> c -> d) ->
      String -> a -> b -> c -> Obs d
...
pat0 :: (a -> Maybe ()) -> Pat a b b
pat1 :: (a -> Maybe b) -> Pat b e f -> Pat a e f
pat2 :: (a -> Maybe (b,c)) -> Pat b e f -> Pat c f g -> Pat a e g
pat3 :: (a -> Maybe (b,c,d)) -> Pat b e f -> Pat c f g -> Pat d g h ->
        Pat a e h
...
patContext :: (a -> [(Int,a)]) -> Pat a b c -> Pat a b c
```

Fig. 1. Combinators for defining patterns for new types

4.4 Deactivating Assertions

Any system supporting assertions enables the programmer to easily deactivate assertions. Hence we provide a module `AssertWithoutCheck` with a function `assert` that is just implemented as the identity function on its third argument and does not check any assertion. To deactivate assertions the programmer replaces `import Assert` by `import AssertWithoutCheck` in their program.

While it may be advisable to leave simple assertions ("argument greater zero") in production code, our pattern logic encourages the formulation of properties of large data structures. Testing these properties is inherently time consuming. For example, it is infeasible in practice to check in a compiler after every update of the symbol table that the whole table is sorted with respect to a key.

5 Defining new Patterns

Using our library does not come for free. The user has to define pattern constructors for their own data types. For each algebraic data type they usually have to define – a context pattern,
– pattern constructors for all its constructors, and
– an instance of the class `Observe`.

To make these definitions as simple as possible, we provide a set of combinators, shown in Figure 1. The implementation of observers and the abstract data type `Obs a`, will be discussed in more detail in Section 6. Here we concentrate on what a programmer has to do to assert properties for their data types. As an example, we introduce a data type `Tree` for polymorphic trees in Figure 2 and show the definitions the programmer has to write for the pattern logic.

First, the programmer has to define an instance of the class `Observe`: for each constructor, they have to define an observation function. We provide generic

```
data Tree a = Node (Tree a) a (Tree a) | Empty

instance Observe a => Observe (Tree a) where
  observe (Node lt n rt) = o3 Node "Node" lt n rt
  observe Empty = o0 Empty "Empty"

pNode :: Pat (Tree a) b c -> Pat a c d -> Pat (Tree a) d e
         -> Pat (Tree a) b e
pNode = pat3 (\t -> case t of Node tl n tr -> Just (tl,n,tr)
                              _            -> Nothing)

pEmpty :: Pat (Tree a) b b
pEmpty = pat0 (\t -> case t of Empty -> Just ()
                               _     -> Nothing)

pTreeC :: Pat (Tree a) b c -> Pat (Tree a) b c
pTreeC = patContext (\t -> case t of Node tl n tr -> [(0,tl),(2,tr)]
                                     Empty -> [])
```

Fig. 2. Extending the pattern logic for polymorphic trees

observers for constructors of any reasonable arity. These observers have to be applied to the constructor function itself, a string representation of the constructor and the arguments obtained from pattern matching. The programmer also has to define the pattern constructors. Again, we provide generic versions for pattern constructors (`pat`n) for each arity. The only argument of these generic patterns is a function which makes pattern matching a total function by means of a `Maybe` type and a tuple of the same arity as the constructor. Finally, the programmer has to define the context pattern for their new type. They should use the generic function `patContext`, which takes a function that determines all arguments in which the type is recursive. We encode these arguments as a list of the argument number and the corresponding actual argument. Note, that descending within a data type only makes sense for arguments of the same data type. Whenever we want to descend another type, we have to add a context of this type. For instance, consider a tree of lists of `Int`s. An arbitrary `Int` within this tree can be selected by the pattern

```
pTreeC (pNode p_ (pListC (val <:> p_)) p_)
```

Although a user has to generate some boilerplate code, we minimised the required amount of work and possible mistakes by defining the abstractions `o`n, `pat`n and `patContext`. In practice the effort for introducing observers and patterns for each user defined datatype should be small. For GHC users we additionally provide a module that enables fully automatic derivation of such instances and functions by means of Template Haskell [14].

5.1 Example: Clausify

The program `clausify` by Colin Runciman takes a propositional formula of type

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

and transforms it into clausal form. The program is a composition of several simple transformation stages. After each successive stage, the following properties should hold, cumulatively:

1. `neg . exists $ pPropC (pImp p_ p_ ||| pEqv p_ p_)`
   Implication (`Imp`) and equivalence (`Eqv`) have been eliminated.
2. `forAll (pPropC (exists (pNeg p_) ==> exists (pNeg (pSym p_))))`
   `Neg (Sym _)` is the only permitted form of negation. Note, that `==>` matches its argument patterns against the same data.
3. `neg . exists $ pPropC (pDis (pCon p_ p_) p_ |||`
   `pDis p_ (pCon p_ p_))`
   No conjunction occurs within a disjunction.

Intentionally introduced faults usually cause the program to abort with a pattern-match failure at a later stage. Our assertions always report a failed assertion before such a pattern-match failure occurs, in contrast to [2], where the same properties are asserted.

6 Implementation

We have space to give only a rough outline of how our assertion library works. The implementation combines two ideas: We use the technique of the Haskell Object Observation Debugger (HOOD) [6] to observe when a part of a value is demanded and get access to this part of the value. We check assertion patterns in coroutines that are implemented via continuations.

We have to check an assertion pattern for the argument of `assert` before the argument is used by the context of the assertion application, but we cannot evaluate the argument further than the context of the assertion application demands. So we use the technique of HOOD and wrap the argument with a function `observe :: Observe a => a -> EvalTreeRef -> a`. This function is a non-strict identity, except that as a side-effect it records how far the value of the argument has been demanded by the context. This information is recorded in an evaluation tree[1]:

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

Every time the argument is further evaluated, the evaluation tree grows at a leaf via a mutable variable `EvalTreeRef`.

All suspended assertions (pattern matches) are stored as `IO` actions in unevaluated leaves. When the corresponding part is evaluated, the `IO` actions are executed and thus checking continues, as the non-empty list case of the `Observe` instance declaration for lists in Figure 3 shows. Checking an assertion pattern is performed on the evaluation tree. The checking functions are defined in continuation style. So when a checking function comes across a leaf of the tree that indicates a yet unevaluated part, the `IO` action is extended by further checks (which themselves can again extend other actions when executed). So we have implemented a scheduler for coroutines with waiting coroutines stored in the evaluation tree. Assuming that the predicates used in patterns terminate, all pattern checking terminates and hence we do not need preemptive concurrency

[1] Hence, the type `Obs` already used in Figure 1 can be defined as:
`type Obs a = EvalTreeRef -> a`

```
instance Observe a => Observe [a] where
  observe [] r = ...
  observe (x:xs) r = unsafePerformIO $ do
    Uneval routines <- readIORef r
    rx <- newIORef (Uneval (return ()))
    rxs <- newIORef (Uneval (return ()))
    writeIORef r (Cons "(:)" [rx,rxs])
    routines                          -- activate suspended assertions
    return (observe x rx : observe xs rxs)
```

Fig. 3. Observe instance for lists

but cooperating coroutines suffice. The `observe` function ensures that all checking coroutines run before a part of an argument is returned to the program context. Thus an assertion will always report failure before a faulty value is returned to the program context.

To illustrate the mechanism of extending suspended checks in more detail, we briefly discuss the code of some patterns. The type `Pat` is defined as a function with result type `IO()` which is stored in the initially unevaluated `EvalTreeRef` within `assert`[2]:

```
type Join = Bool -> IO ()
type Pat a b c = Bool -> EvalTreeRef -> a -> b -> Join ->
                 (Join -> c -> IO ()) -> IO ()
```

Successively, the arguments of the `Pat` function have the following meanings

- a Boolean value distinguishing the two quantification contexts,
- the evaluation tree on which the pattern is supposed to be checked,
- the real value (which may not be evaluated further),
- the partially applied check function,
- a join function which combines results of parallel pattern matching by means of `(&&&)` or `(|||)` and which depends on quantification,
- and a continuation passing a join function and the remaining checks (`c`) to be performed in `val` patterns.

The pattern matching itself can be illustrated with the definition of `(<:>)`:

```
(<:>) :: Pat a b c -> Pat [a] c d -> Pat [a] b d
(patx <:> patxs) ex r y p join c = do
  evalT <- readIORef r
  case evalT of
   Uneval routines ->       -- extend suspended assertions
     writeIORef r (Uneval (routines >> patx <:> patxs ex r y p join c))
   Cons _ rs -> case y of
                  (y:ys) -> let [rx,rxs] = rs in
                            patx ex rx x p join
                              (\join2 p2 -> patxs ex rxs xs p2 join2 c)
                  _      -> join (not ex)
```

[2] To obtain better type error messages, `Pat` is defined as an abstract datatype guarded by a constructor in the real implementation.

If the data structure to be matched has not been evaluated yet, then the suspended check action is extended with the actual matching. Otherwise, pattern matching is performed. If it succeeds, the sub-patterns are matched (in continuation style). If it fails, no further pattern matches have to be performed; the local result of pattern matching can be combined with other pattern matches executed in "parallel", which should become clearer from the definition of the parallel conjunction and disjunction of patterns. Both can be defined by means of a more general function `(***)` which stores two patterns within the evaluation tree:

```
(***) :: Pat a b c -> Pat a b c -> Pat a b c
(***) pat1 pat2 ex r x p join c = do
  rcount <- newIORef 2
  pat1 ex r x p (newJoin rcount join) c
  pat2 ex r x p (newJoin rcount join) c
  where
    newJoin = if ex then ... else ...

forAll :: Pat a b c -> Pat a b c
forAll pat _ = pat False

exists :: Pat a b c -> Pat a b c
exists pat _ = pat True

(&&&) :: Pat a -> Pat a -> Pat a
pat1 &&& pat2 = forAll (pat1 *** pat2)

(|||) :: Pat a -> Pat a -> Pat a
pat1 ||| pat2 = exists (pat1 *** pat2)
```

The initial value of `join` is a function that prints that the assertion succeeded or failed, depending on the Boolean value. In the definition of the combinator `(***)` the `join` continuation is extended. The `newJoin` applied in both coroutines uses a common reference so that a coroutine can determine if it is the first to do the join. Thus a parallel conjunction or disjunction can be implemented. When the first coroutine yields `False` for an argument of a conjunction, the result of the conjunction is determined and the coroutine evaluates the remaining `join`. When the first coroutine yields `True` for an argument of a conjunction, it updates the common reference accordingly and terminates. The second coroutine will evaluate the remaining `join`. We obtain a parallel implementation of conjunction and disjunction.

Unlike [2] our implementation of assertions does not use any features of Concurrent Haskell [12], which is a substantial extension of Haskell that is fully implemented only in GHC. Like [2] we need references to mutable variables in the IO monad (`IORefs`) and the function `unsafePerformIO :: IO a -> a`. These two language extension are provided by all Haskell systems. Using the function `unsafePerformIO` is dangerous, because it bypasses the safety net of the type system. The alternative would be to modify a compiler and its run-time system, which would be non-portable and far more complex.

7 Related Work

Assertions have been used in programs since the 1970s [11, 13] and are directly supported by many programming languages. In particular the object-oriented programming language Eiffel is based on a "Design by Contract" philosophy and language constructs directly support assertions to express contracts [8, 9].

Assertion-based contracts have been introduced into the Scheme community [5]. Findler and Felleisen motivate how assertions enable the programmer to express interesting properties that they cannot express in existing type systems. The two issues dealt with here, ensuring that assertions are both lazy and prompt, do not arise for a strict language such as Scheme, because all expressions are fully evaluated before an asserted property needs to be checked. Hence arbitrary Scheme expressions can be used to express properties, but a pattern logic might increase the usability of such contracts as well. Properties of functions are also expressed as properties of the argument-result pairs. A major concern of [5] is to determine which part of a program has to be blamed for the failure of an assertion. For example, when the pre-condition of a function fails, the caller of the function is to blame, when the post-condition fails, the function itself is to blame. Because in a lazily evaluated language the runtime stack does not reflect the call-structure, assigning blame is more complex. A cooperation with the Haskell tracer Hat [4] and its redex trail view may provide a solution in the future. Recently the contracts for Scheme have been transferred to Haskell [7], but without taking account of its lazy semantics.

Chitil, McNeill and Runciman [2] previously expressed the need for assertions to be lazy in a lazy language. They give several implementations but because properties are expressed as arbitrary Boolean-valued functions, assertions are not prompt but often get stuck. Their most advanced implementation requires Concurrent Haskell and their synchronisation that gives assertion threads higher priority than the main computation can cause deadlocks. Properties of functions can only be asserted by a special assertion combinator for functions with limited expressibility. We can define a similar combinator with our pattern logic as well:

```
assertFun :: (Observe a, Observe b) => String ->
             Pat a c d -> Pat b d Bool -> c -> (a -> b) -> (a -> b)
assertFun label patA patB p fun a = b'
  where (a',b') = assert label (check (pPair patA patB) p) (a,fun a')
```

To make assertions lazy, we have adapted the lazy observation technique of the Haskell Object Observation Debugger (HOOD) [6]. In every application area we know of it is used slightly differently. So the original HOOD records a linear trace of events, in [2] a copy of the observed value is recorded, and here we record the evaluation tree. We also intimately link scheduling of coroutines with observations. Nonetheless we believe that it is possible to wrap up this useful technique once and for all in a library that can then be used for the listed and future application areas.

There are numerous proposals in the literature for extending the pattern matching facilities of functional programming languages. Our context pattern combinators were inspired by [10] and the pattern logic is similar to regular

expressions [1]. All these proposals aim to extend the expressiveness of the programming language and the semantics of extended patterns is similar to that of normal patterns. The pattern logic for prompt lazy assertions requires a different semantics. Previous papers propose language extensions that require compiler modifications or preprocessors, whereas we provide a portable library. Also, each of our context pattern combinators matches only the data constructors of a single type. Thus they are more specific and easier to use.

Basically our patterns describe a grammar and our pattern combinators are parser combinators [15]. They do not parse a string or list of tokens but tree-structured data. Hence our combinators have to leave the sequential structure of normal parser combinators. As grammars describe context-free properties, the similarity to parsing combinators gives an indication of the expressiveness of our pattern logic; however, the combinator `check` used with several arguments goes beyond grammars and allows the specification of context-sensitive properties.

QuickCheck is a library for testing Haskell programs with random data [3]. Properties are expressed as Haskell functions. For example, the property that the function `insert` preserves order can be expressed as follows:

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

QuickCheck properties can use normal pattern matching and Boolean functions, because they are only checked for total, finite data structures that are randomly generated. Testing with random data and testing with real data as our assertions do are two different methods which complement each other. A combined tool is feasible, but to handle laziness it would need to use our pattern logic.

QuickCheck and assertions handle preconditions (like `sorted xs` in the example) in distinct ways. In QuickCheck a precondition is a filter on the test data, so that a strong precondition makes it hard to obtain a sufficient amount of test data. When that is the case, the user is left with the difficult task of defining a special test generator that generates data fulfilling the precondition (for example generate sensible abstract syntax trees for testing compiler phases). In contrast, our assertions check for every call of a function that its preconditions are met by the caller. So assertions naturally support the contract between caller and callee whereas for QuickCheck preconditions cause additional problems.

8 Conclusions

We have presented a new approach for assertions in lazy functional programming languages such as Haskell. Our assertions do not modify the run-time behaviour of lazy execution (unless a predicate used by `check` fails to terminate). Assertions are implemented by means of a pattern logic, a high level, abstract specification language. Assertions provide a parallel implementation of conjunction and disjunction, which makes it possible to report failure of assertions promptly, before faulty values can effect the rest of the computation. Our approach is implemented as a library, without any modification of the compiler or the run-time system, and only needs common extensions of Haskell 98.

For future work, we plan to add assertions to more real-life programs. The practical experience we will gain will guide us in improving our pattern logic. We may revise some design decisions and possibly add further combinators to make the logic more expressive and/or easier to use.

We will also investigate which function should be blamed when an assertion fails. Combining assertions with the Haskell tracer Hat [4] should enable the programmer to locate the function to blame and even the precise fault location.

References

1. N. Broberg, A. Farre, and J. Svenningsson. Regular expression patterns. In *ICFP '04: Proceedings of the ninth ACM SIGPLAN international conference on Functional programming*, pages 67–78, New York, NY, USA, 2004. ACM Press.
2. O. Chitil, D. McNeill, and C. Runciman. Lazy assertions. In P. Trinder, G. Michaelson, and R. 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. K. Claessen, C. Runciman, O. Chitil, J. Hughes, and M. Wallace. Testing and Tracing Lazy Functional Programs using QuickCheck and Hat. In *4th Summer School in Advanced Functional Programming*, LNCS 2638, pages 59–99, August 2003.
5. R. B. Findler and M. 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. R. Hinze, J. Jeuring, and A. 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. B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.
9. B. Meyer. *Eiffel: The Language*. Prentice-Hall, Inc., 1992.
10. M. Mohnen. Context patterns, part II. In *Implementation of Functional Languages*, LNCS 1467, pages 338–357, 1997.
11. D. L. Parnas. A technique for software module specification with examples. *Commun. ACM*, 15(5):330–336, 1972.
12. S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Conference Record of POPL '96: The* 23rd *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 295–308, 21–24 Jan. 1996.
13. D. S. Rosenblum. A practical approach to programming with assertions. *IEEE Trans. Softw. Eng.*, 21(1):19–31, 1995.
14. T. Sheard and S. P. Jones. Template metaprogramming for haskell. In *Haskell Workshop 2002*, October 2002.
15. Swierstra and Alcocer. Fast, error correcting parser combinators: A short tutorial. In *Theory and Practice of Informatics, Seminar on Current Trends in Theory and Practice of Informatics, LNCS*, volume 1725. 1999.