Lazy and Enforceable Assertions for Functional Logic Programs

Michael Hanus

Institut für Informatik, CAU Kiel, D-24098 Kiel, Germany
mh@informatik.uni-kiel.de

Abstract. Assertions or contracts are an important technique to improve the quality of software. Thus, assertions are also desirable for functional logic programming. Unfortunately, there is no established meaning of assertions in languages with a demand-driven evaluation strategy. Strict assertions are immediately checked but may influence the behavior of programs. Lazy assertions do not modify the behavior but may not be faithful since some assertions might not be checked at all. In order to avoid the disadvantages of strict and lazy assertions, we propose enforceable assertions that are delayed as lazy assertions but can be explicitly enforced at some point where faith is required, e.g., at the end of the program execution or before irrevocable I/O actions. We describe a prototypical implementation of this idea in the functional logic language Curry where the programmer can choose between lazy and enforceable assertions.

1 Motivation

The use of assertions or contracts is an important technique to improve the quality of software [21]. Assertions make certain assumptions in the code explicit, e.g., requirements on argument values to ensure the correct execution of a function's code. In principle, assertions can be implemented in the program's code by including code to check them. For instance, one can raise an exception if the factorial function is called with a negative argument or the `head` function is applied to an empty list. In order to keep the application code comprehensible and maintainable, it is preferable to have a clear distinction between application code and assertions so that one can later decide how to treat assertions. For instance, they can be always checked, checked only during the development and test of the application program, or removed after proving that they hold in the application program.

Design by contract has been introduced in the context of object-oriented programming [21]. It is also reasonable to use it in other programming paradigms. In this paper we consider the inclusion of assertions[1] in functional logic programs.

[1] We use the term "assertions" for properties of values, whereas "contracts" are used for properties of operations consisting of assertions for arguments as well as assertions for result values.

Thus, we assume familiarity with basic concepts of functional logic programming (details can be found in recent surveys [4,15]). For our examples and implementation, we use the declarative multi-paradigm language Curry [12,17] that combines functional programming features (demand-driven evaluation, higher-order functions) with logic programming features (computing with partial information, unification, non-deterministic search for solutions) and concurrent evaluation. The syntax of Curry is almost identical to Haskell [22]. In addition to Haskell, Curry allows the declaration of free (logic) variables by the keyword "`free`".

The intuitive meaning of assertions is as follows. If we decorate an expression in a program with an assertion, e.g., a predicate, then an assertion violation should be reported whenever, during the program's execution, the expression has some value that does not satisfy the assertion. Unfortunately, a precise definition and implementation of assertions is not straightforward in the context of functional logic programming due to the demand-driven evaluation strategy. This problem is already present in non-strict functional languages where various proposals have been made to tackle it (e.g., [6,7,8]). In order to discuss the difficulties in more detail, consider a simple approach to introduce assertions in a functional (logic) language by defining a combinator that attaches an assertion to an expression:

```
assert :: (a → Bool) → a → a
assert p x = if p x then x
                    else error "Assertion failed"
```

Here, the assertion is a predicate on the values of the expression. If the predicate applied to the expression evaluates to `True`, the expression is returned, otherwise an exception is raised. Since this definition has the effect that the assertion is immediately checked, we call such an assertion also *strict assertion*.

A disadvantage of strict assertions is the fact that they are not *meaning preserving*, i.e., they might influence the behavior of application code, even if all assertions are satisfied. For instance, consider an assertion that states that a list is ordered:

```
ordered []       = True
ordered [_]      = True
ordered (x:y:ys) = x<=y && ordered (y:ys)
```

Then the evaluation of "`head (assert ordered [1,2..])`" does not terminate due to the evaluation of the infinite list argument caused by the assertion.

To avoid this influence of assertions to the application program, Chitil et al. [8] proposed *lazy assertions* that do not enforce argument evaluation but are checked when the argument expression has been evaluated by the application program so far that the assertion can be evaluated without further evaluation of its argument. Thus, as long as all assertions are satisfied, program executions with or without lazy assertion checking deliver the same results. A disadvantage of lazy assertions is the fact that some obviously violated assertions are not reported when the arguments are not sufficiently evaluated. For instance, "`head`

`(assert ordered [2,1])`" returns `2` without any assertion violation if the assertion is lazily checked, although the programmer assumes, due to the intuitive meaning of assertions, that the result is the minimal element of the list.

Chitil and Huch [6,7] improved the situation by introducing a specific assertion language that supports assertions where some violations can be earlier reported in comparison to lazy assertions. Nevertheless, the basic problem remains: it is possible that the violation of assertions might be undetected or detected too late if all assertions are lazily checked. One can argue that this behavior is fine since the possibly unchecked parts of an expression are not necessary for computing the result. However, this view changes when I/O actions are taken into account. For instance, if we pass a data structure through a sequence of I/O actions where a first action needs only some part of the structure (similarly to the call to `head` above) and the other parts are needed by subsequent actions, then a violation of an assertion for the data structure might be detected too late, e.g., after some rocket has been launched. Thus, there are situations where one wants to ensure that the assertions hold. Degen et al. [9] put this into the slogan "faithfulness is better than laziness."

Altogether, there is no silver bullet for assertions in non-strict languages since lazy assertions might not be faithful, and strict assertions cannot be applied to algorithms exploiting infinite data structures. Thus, we propose an intermediate approach where the user can choose between lazy and faithful assertions. In contrast to strict assertions, we propose to evaluate even faithful assertions in a lazy manner but enforce their evaluation at particular program points, e.g., before I/O actions where faith is important or at the end of the program execution. Therefore, we call this kind of assertions "enforceable."

One might wonder why it is useful to evaluate faithful assertions not simply as strict assertions but treat them in a lazy manner. The reason is that this strategy can reduce the possibility to report false violations of assertions, i.e., violations that do not occur during the execution of the application program. Note that lazy evaluation in functional *logic* programs is not only desirable to obtain optimal evaluation [2] but also to reduce the search space, i.e., lazy evaluation on non-deterministic programs yields a demand-driven exploration of the search space (see [1,15]). As a consequence, it is reasonable to evaluate expressions lazily even if we know that we want to evaluate them completely. For instance, consider the following program:

```
f 0 = 0
f 1 = 1

g x = [f x]

h False 1 = 2
h True  0 = 3

null []    = True
null (_:_) = False
```

The main expression to be evaluated is

```
let x free in h (null (g x)) x
```

Functional logic languages with a lazy evaluation strategy, like Curry [12,17] and $\mathcal{TOY}$ [19], evaluate this expression to `2` by instantiating `x` to `1` (after evaluating `(null (g x))` to `False`). Now consider that we put a post-condition on `g` ensuring that all values in the result list are positive whenever we apply `g` to some argument during the program execution, i.e., we modify the definition of `g` to

```
g x = assert (all (>0)) [f x]
```

Remember that the evaluation of the main expression requires the evaluation of `(g x)` to an empty or non-empty list. Thus, if the assertion on the result of `g` is eagerly evaluated, the element `f x` in the result list is evaluated to `0` by instantiating `x` to `0`. Hence, an exception is raised due to the violated assertion. However, if we delay the assertion checking until the end of the regular program execution (which instantiates `x` to `1`), no exception will be raised since the assertion is satisfied.

Thus, we propose in this paper an assertion framework for functional logic programs with the following characteristics:

- The programmer can put lazy or enforceable assertions on expressions in the application program.
- Lazy assertions are checked when the arguments of the assertions are evaluated by the application program so that the assertions can be reduced to a Boolean value without any further evaluation of the arguments. Thus, lazy assertions do not initiate any argument evaluation by themselves.
- Enforceable assertions are also lazily checked, i.e., when the values of their arguments are available, they are reduced to a Boolean value. In addition, the programmer can specify execution points (I/O actions) where all enforceable assertions are eagerly checked if they have not already been checked.
- We describe a prototypical implementation of the framework in Curry. The implementation is defined as a library implemented in Curry based on a few extensions available in the Curry implementation PAKCS [16].

The next section defines the kind of assertions which we propose in this paper. Section 3 describes an implementation of our assertion concept in Curry. Section 4 discusses some related work before we conclude in Section 5.

2 Assertions

As discussed in Section 1, assertions can be considered as predicates on expressions, i.e., they are of type "`a -> Bool`" where `a` is the type of the considered expression. Of course, one can deal with richer and more specialized assertion languages, like [6,7,18], but this is outside the scope of this paper. Therefore, we simply consider assertions as standard predicates. As shown in the previous section, one could attach an assertion to an expression by a combinator

```
assert :: (a → Bool) → a → a
```

However, this is not sufficient for an implementation of lazy assertions since they need to inspect the data on which they operate. Therefore, they are not parametrically polymorphic but their behavior depends on the structure of the concrete type. Hence, we adopt a technique used in observation debugging tools for functional (logic) languages [5,11] and put some information about the considered types as an additional argument of type "`Assert a`".[2] Furthermore, we also add a string argument that is used to identify the violated assertion when an exception is raised.[3] Altogether, the assertion combinator has the following type:

```
assert :: Assert a → String → (a → Bool) → a → a
```

In order to attach concrete assertions to a program, our assertion library defines constants and functions returning assertion information for particular types, like

```
aInt   :: Assert Int
aFloat :: Assert Float
aChar  :: Assert Char
⋮
aList  :: Assert a → Assert [a]
aPair  :: Assert a → Assert b → Assert (a,b)
⋮
```

A concrete assertion is defined by combining these operations in a type correct way. For instance, the post-condition on the operation `g` as shown in Section 1 can now be defined by

```
g x = assert (aList aInt) "AllPositive" (all (>0)) [f x]        (1)
```

These assertions are *enforceable*, i.e., they are lazily evaluated with the same demand as the application program but they can also be eagerly checked at particular program points. For the latter purpose, the library defines an I/O action

```
enforceAssertions :: IO ()
```

that forces the evaluating of all pending assertions. Thus, one can check all assertions at the end of the user program (`main`) by executing

```
main >> enforceAssertions
```

---

[2] In Haskell one could add the `Assert` information via type classes, but type classes are not yet included in Curry.

[3] Of course, it would be better to show the position of the violated assertion in the exception. Since this information cannot be obtained by a library but requires specific compiler support, we omit it here. In a future version, the compiler might introduce the position information in the string argument.

Of course, `enforceAssertions` can also be used any number of times during the regular program execution, e.g., before "important" I/O actions of the user program.[4]

If the programmer wants to define assertions that should be only lazily evaluated (i.e., they might not be faithful but could be desirable for assertions on infinite data structures), our library also provides an operator to attach such *lazy assertions* to expressions:

```
assertLazy :: Assert a → String → (a → Bool) → a → a
```

In order to test and compare the various assertion methods, our library also defines a *strict assertion*, which is immediately checked, by

```
assertStrict :: Assert a → String → (a → Bool) → a → a
assertStrict _ id p x =
   if p x then x
          else error ("Strict assertion '"++id++"' failed!")
```

Consider again the example given in Section 1 where the function `g` is defined with a post-condition as shown in program rule (1) above. Our implementation, described in detail below, evaluates the expression "`h (null (g x)) x`" without reporting an assertion violation, as intended. However, if we replace in rule (1) "`assert`" by "`assertStrict`", the evaluation of the same expression yields an exception reporting that the assertion `AllPositive` is violated, which is not true in the program executed without assertions. This example shows the usefulness to evaluate enforceable assertions in a lazy manner. The next section shows an implementation of this concept in Curry.

3 Implementation

In this section, we first describe an implementation of purely lazy assertions in Curry. Based on this, we develop an implementation of enforceable assertions.

3.1 Lazy Assertions

A possible implementation of lazy assertions in Haskell has been proposed in [8]. Our implementation[5] uses similar ideas but is based on functional logic programming features. Lazy assertions should only be checked when the application program demands and evaluates the arguments of the assertions. In order

[4] In principle, one could also enforce assertion checking at any program point by wrapping `enforceAssertions` using `unsafePerformIO`. However, it would be unclear when and whether these assertions will be checked due to the overall lazy evaluation strategy. Therefore, we prefer to support only I/O actions to enforce assertion checking.

[5] This implementation is partially based on code developed with Bernd Braßel and Olaf Chitil.

to avoid the evaluation of arguments by assertion checking, we wrap these arguments with a function `wait` that is evaluable only if the original argument has been evaluated by the application program. In [8] this is implemented via concurrent threads which synchronize on `IORef`s. Since Curry subsumes the concept of concurrent logic programming, we use these features to implement the concurrent evaluation of lazy assertions.

To implement lazy assertions, we require for each concrete type τ two operations:

```
wait    :: τ  → τ
ddunify :: τ  → τ  → τ
```

`wait` is the identity function on values of type τ but suspends as long as the value is not provided, i.e., an unbound variable.[6] For instance, consider the type `Nat` of natural numbers in Peano's notation defined by

```
data Nat = Z | S Nat
```

The corresponding operation `waitNat` can be defined as follows:

```
waitNat :: Nat  → Nat
waitNat x = case x of Z    → Z
                      S y  → S (waitNat y)
```

Clearly, `waitNat` is the identity on `Nat` values but suspends when it is applied to a free variable (due to the `case` construct which suspends on free variables, see [17]).

The second important operation, `ddunify`, implements a demand-driven unification of its arguments. Conceptually, a call "`ddunify x e`" evaluates `e` (demand-driven, i.e., to head-normal form), returns the result, and unifies `x` (which is usually a free variable) with the result's top-level constructor and recurs on the arguments. For instance, the corresponding operation for the type `Nat` is defined as follows:

```
ddunifyNat :: Nat  → Nat  → Nat
ddunifyNat x e =
 if isVar e then (x=:=e) &> e
            else case e of
                   Z    → (x =:= Z) &> Z
                   S y  → let z free
                          in (x =:= S z) &> S (ddunifyNat z y)
```

The test function `isVar` checks whether the current argument evaluates to a free variable (note that free variables are also head-normal forms in functional logic programs). Although this test function is non-declarative (it has been introduced in [5] for a similar purpose), it is necessary to check the state of the argument in order to avoid its unintended instantiation. If the argument `e` evaluates to a free

[6] The suspension is important to ensure the principle that lazy assertions should not change the evaluation behavior of the application program.

variable, it is unified (by the equational constraint "`=:=`") with the argument `x` and returned.[7] Otherwise, the possible constructor-rooted values are examined. In case of the constructor `Z`, this constructor is unified with argument `x` and returned. If the argument's value is rooted by the constructor `S`, `x` is instantiated to the constructor `S` with a fresh variable argument and the corresponding arguments are further unified by `ddunifyNat`.

As we will see below, the operations `wait` and `ddunify` are sufficient to implement lazy assertions. Thus, the type `Assert` to encapsulate type-specific assertion information is defined as

```
data Assert a = Assert (a → a) (a → a → a)
```

where the first and second component are the type-specific `wait` and `ddunify` operations, respectively. Thus, an instance of `Assert` for the concrete type `Nat` can be defined by

```
aNat :: Assert Nat
aNat = Assert waitNat ddunifyNat
```

Based on this structure of the type `Assert`, we can implement the combinator `assertLazy` by applying the operations passed with the `Assert` argument:

```
assertLazy :: Assert a → String → (a → Bool) → a → a
assertLazy (Assert wait ddunify) label p e =
  spawnConstraint (check label (p (wait x))) (ddunify x e)
 where x free
```

The operation `spawnConstraint` (first introduced in [5]) is identical to the guarded expression operator "`&>`" from a declarative point of view. In contrast to "`&>`", `spawnConstraint` proceeds with the evaluation of the second argument even if the evaluation of the guard (first argument) suspends (i.e., the guard is concurrently evaluated). Thus, `assertLazy` evaluates its expression argument `e` and unifies its value with the free variable `x` in a demand-driven manner (by "`ddunify x e`"). Concurrently, the assertion `p` is applied to `x` wrapped by the operation `wait` in order to delay the evaluation until the required argument value is available. The operation `check` simply examines the result of assertion checking and raises an exception, if necessary:

```
check :: String → Bool → Success
check label result =
  case result of
    True  → success
    False → error ("Lazy assertion '"++label++"' violated!")
```

Due to the use of the operations `wait` and `ddunify`, the behavior of the computation of the application program is not changed by attaching lazy assertions to expressions. Since spawned constraints are evaluated with a high priority, a

[7] "`c &> e`" denotes a *guarded expression* where the infix operator is predefined by the conditional rule "`c &> e | c = e`".

violated assertion is reported as soon as it can be decided by the availability of the argument values.

Before we discuss the implementation of enforceable assertions, we show further instances of `Assert` for some concrete types. First we note that the initial test for the free variable case in `ddunify` is necessary for any concrete type. Therefore, we define a generic combinator which adds this test to an arbitrary demand-driven unification:

```
withVarCheck :: (a → a → a) → a → a → a
withVarCheck ddunif x e = if isVar e then (x=:=e) &> e
                                     else ddunif x e
```

To avoid the application of this combinator to each new `Assert` instance, we make the type `Assert` abstract, i.e., we do not export its data constructor but define an operation that constructs an `Assert` instance from a given `wait` and `ddunify` operation:

```
makeAssert :: (a → a) → (a → a → a) → Assert a
makeAssert wait ddunif = Assert wait (withVarCheck ddunif)
```

Using these auxiliary operations, the definition of an `Assert` instance for a concrete type amounts to a simple case distinction on the data constructors of this type. For instance, the instance `aNat` already shown above can also be defined as follows:

```
aNat = makeAssert waitNat ddunifyNat
 where waitNat x = case x of Z    → Z
                             S y  → S (waitNat y)

       ddunifyNat x e = case e of
                    Z    → (x =:= Z) &> Z
                    S y  → let z free
                           in (x =:= S z) &> S (ddunifyNat z y)
```

Instances of `Assert` for polymorphic type constructors are functions parameterized with assertion information related to their type parameters. For instance, the combinator `aList` can be defined as follows:

```
aList :: Assert a → Assert [a]
aList (Assert waita ddunifya) = makeAssert waitList ddunifyList
 where
  waitList l = case l of
                 []     → []
                 (x:xs) → waita x : waitList xs

  ddunifyList x e = case e of
    []   → (x=:=[]) &> []
    y:ys → let z,zs free
           in (x=:=z:zs) &> (ddunifya z y : ddunifyList zs ys)
```

One can use this combinator to define `Assert` instances for specific list types. For instance, the `Assert` instance for strings can be defined as:

```
aString :: Assert String
aString = aList aChar
```

Here we use the `Assert` instance, `aChar`, for characters. To define it and similar instances for primitive types (i.e., unstructured types that have only simple constants as values), we define a generic instance for primitive types:

```
aPrimType :: Assert a
aPrimType = Assert waitPrimType ddunifyPrimType
  where waitPrimType = ensureNotFree
        ddunifyPrimType x i | x =:= i = i
```

The predefined operation `ensureNotFree` returns its argument evaluated to head-normal form but suspends as long as the result is a free variable. Hence, the application of `ensureNotFree` avoids an unintended instantiation of free variable arguments during the evaluation of an assertion. Now, we can define `Assert` instances for various primitive types:

```
aInt :: Assert Int
aInt = aPrimType

aChar :: Assert Char
aChar = aPrimType

⋮
```

We have shown the implementation of `Assert` instances for a few types. Note, however, that the code structure of these instances follows a scheme which depends on the structure of the type definitions. Thus, it is easy to provide a tool to mechanically generate these instances for any user-defined data type.

3.2 Enforceable Assertions

As mentioned above, enforceable assertions are evaluated like lazy assertions during the standard execution of the application program. In addition, they are eagerly evaluated when it is requested by the I/O action `enforceAssertions`. This demands for two execution modes of enforceable assertions:

1. demand-driven evaluation like lazy assertions, and
2. eager evaluation like strict assertions.

The implementation of lazy assertion wraps the arguments via `wait` operations as shown above, i.e., their values are not available to the assertion as long as they are not demanded by the application program. Thus, there seems to be no way to enforce the evaluation of a lazy assertion outside the application program. Due to this consideration, we implement the eager evaluation mode of an enforceable assertion by creating a further application of the assertion to its original

arguments. The evaluation of this application is delayed until it is requested by `enforceAssertions`. Obviously, this scheme could cause some re-evaluation of the assertion when the evaluation of the lazy assertion has already been started before `enforceAssertions` occurs. However, this is not a serious problem since the evaluation of the arguments are shared (due to the lazy strategy of the host language) and under the assumption that assertion evaluation has not a high complexity (which should be satisfied in order to execute programs with assertion checking in a reasonable amount of time).

At least, we can avoid the re-evaluation of an already evaluated assertion by passing a free variable as an "evaluation flag." Thus, we extend the implementation of lazy assertions shown in Section 3.1 to enforceable assertions as follows:

```
assert :: Assert a → String → (a → Bool) → a → a
assert (Assert wait ddunify) label p e
  | registerAssertion eflag label (p e)
  = spawnConstraint (check label eflag (p (wait x)))
                    (ddunify x e)
 where eflag,x free
```

The operation `registerAssertion` suspends the evaluation of its last argument until it is requested by `enforceAssertions`. The free variable `eflag` is the flag to avoid a complete re-evaluation of the assertion. For this purpose, we redefine the function `check` presented above so that the variable `eflag` is instantiated when the assertion is fully evaluated (the instantiation in the `False` case is reasonable if the exception is caught):

```
check label eflag result =
  case result of
    True  → eflag=:=()
    False → eflag=:=() &>
            error ("Lazy assertion '"++label++"' violated!")
```

Next we discuss the implementation of `registerAssertion`. Suspended computations can be easily obtained in Curry by waiting on the instantiation of a free variable. Since all enforceable assertions should be evaluated if `enforceAssertions` occurs, these suspended computations must share the same free variable. Therefore, we use the Curry library `GlobalVariable`[8] that supports the definition of typed "global variables." A global variable is a top-level entity that has an associated term. Furthermore, the association can be changed by I/O actions. A global variable `g` having associated terms of type τ is defined in a Curry program by

```
g :: GVar τ
g = gvar t
```

[8] `http://www.informatik.uni-kiel.de/~pakcs/lib/CDOC/GlobalVariable.html`

where t is the initially associated term (of type τ). Furthermore, there are two operations

```
readGVar  :: GVar a  → IO a
writeGVar :: GVar a  → a  → IO ()
```

to get and set the term associated with a global variable. Note that the associated term can also contain free variables.

In our implementation of enforceable assertions, we define a global variable that is associated with a free variable used to synchronize all enforceable assertions:[9]

```
assertionControl :: GVar ()
assertionControl = gvar unknown
```

Using this global variable, we can define the operation to register an enforceable assertion for later evaluation as follows:

```
registerAssertion eflag label asrt =
  spawnConstraint (delayedAssertion eflag label asrt =:= ())
                  success

delayedAssertion eflag label asrt = unsafePerformIO $ do
  v <- readGVar assertionControl
  ensureNotFree v =:= ()  &> done -- suspend on control variable
  if isVar eflag && not asrt
   then error ("Enforceable assertion '"++label++"' violated!")
   else done
```

Since global variables are handled by I/O actions, we have to use the operation `unsafePerformIO` to put the execution of an I/O action into a non-I/O value. Hence, `registerAssertion` spawns a constraint which waits on the instantiation of the variable that controls enforceable assertions. If this variable is instantiated, it is checked whether the evaluation flag is instantiated, i.e., whether the corresponding lazy assertion was already evaluated. If this is not the case, the assertion `asrt` is evaluated and an exception is raised in case of a violation.

Now it is easy to request the evaluation of enforceable assertions by instantiating the global control variable:

```
enforceAssertions = do
  v <- readGVar assertionControl
  v=:=() &> done           -- instantiate control variable
  writeGVar assertionControl unknown
```

After the instantiation, a new free variable is associated with the global control variable in order to have it ready for subsequent occurrences of enforceable assertions.

[9] The operation `unknown` is defined by "`unknown = x where x free`" in the Curry prelude, i.e., it returns a free variable.

Note that we used a few non-declarative features to implement lazy and enforceable assertions. This is not a problem for the application programmer since their use is completely hidden in the library implementing this assertion framework. The use of the non-declarative constructs is quite helpful to obtain a maintainable high-level implementation without extending the run-time system of the underlying Curry implementation.

3.3 More Assertions

We can use the assertion implementation to provide some other useful assertions. For instance, an assertion for a function could check whether all pairs of argument/result values satisfy a given predicate:

```
assertFun :: Assert a → Assert b → String → (a →b →Bool)
             → (a →b) → a → b
assertFun (Assert waita ddunifya) (Assert waitb ddunifyb)
          label p f x
  | registerAssertion eflag label (p x (f x))
  = spawnConstraint (check label eflag (p (waita wx) (waitb wfx)))
                    (ddunifyb wfx (f (ddunifya wx x)))
 where eflag,wx,wfx free
```

As an example for the use of this assertion, consider that we want to check whether a function `f` of type `Int → Int` behaves monotonically for all calls. Thus, we wrap it with the assertion

```
assertFun aInt aInt "monotonic" (<) f
```

Contracts [21] are requirements on the argument and result values when operations are invoked. Thus, a contract contains a precondition (on the argument) and a postcondition (relating the argument and the result) for an operation. In order to model such kinds of contracts, we can define a contract as an assertion on the argument and an assertion between argument and result values as follows:

```
contract :: Assert a → Assert b → String
            → (a →Bool) → (a →b →Bool)
            → (a →b) → a → b
contract asrta asrtb label argp funp f x =
  assertFun asrta asrtb ("Result of "++label) funp f
            (assert asrta ("Argument of "++label) argp x)
```

For instance, if we want to turn a function `fac` into a function `cfac` containing the contract ensuring that `fac` is always called with a non-negative argument and returns a positive argument, we define it as follows:

```
cfac = contract aInt aInt "fac" (>=0) (\_ r →r>0) fac
```

Sometimes one is interested to ensure that specific arguments are free variables when they occur for the first time. For instance, there are high-level libraries for

GUI or HTML programming that use free variables as logical references between widgets and event handlers [13,14]. Although these libraries use abstract data types in order to ensure this property, it might be also useful to check this property by an assertion, in particular, during the development of such libraries. This can be done by an assertion that raises an exception when the argument is not a free variable. The implementation is quite easy (note that enforceable assertions are not necessary here since this assertion is immediately checked when it occurs):

```
assertLogVar :: String → a → a
assertLogVar label x = (check label () (isVar x)) &> x
```

4 Related Work

Assertions or contracts have been introduced in the context of imperative object-oriented programming languages [21], but assertions are also useful for declarative programming when larger software systems are developed. Although powerful type systems can express assertions on operations that can be checked at compile time, more complex properties, like orderings on lists or trees, non-negative values etc, cannot be expressed by standard type systems. If application programs become more complex, it is important to state and check such properties, e.g., to improve reliability or to locate the source of program bugs more easily.

Since the demand-driven evaluation model of functional languages causes additional difficulties when considering assertions, there are a number of different proposals in this area. Chitil et al. [8] proposed lazy assertions for non-strict functional languages. They suggested that assertions should not influence the normal behavior of programs (apart from space and time needed for assertion checking). For this purpose, they discussed different implementations. To ensure early detection of assertion violations, they implemented assertions by concurrent threads. Although we used the concurrent logic programming features of Curry to implement assertions, both implementations of lazy assertions have many similarities, in particular, both are based on some non-declarative constructs like `unsafePerformIO`.

Since lazy assertions might not detect assertion violations if parts of the considered data structures are not demanded by the application program, Chitil and Huch [7] improved the situation by introducing a specific pattern logic to express assertions that allow an earlier detection of violated assertions. In particular, they proposed to replace sequential Boolean operators, like "`&&`" or "`||`", by corresponding operators that evaluate their arguments in parallel. Although such an extension could also be interesting for our framework, this does not make our proposal for enforceable assertions superfluous: even in the extended assertion language of Chitil and Huch, it might be the case that some violated assertions remain undetected in a program execution.

Degen et al. [9] discussed the various requirements and possibilities of assertion checking in lazy languages and came to the conclusion that there is no method satisfying all desirable requirements. Thus, one has to choose between meaning preserving (i.e., lazy) or faithful (i.e., strict) assertions. We have shown that in functional logic programming, there is a further interesting approach: enforceable assertions that are not immediately checked but delayed to a point where faith is strictly required.

Hinze et al. [18] introduced a domain-specific language for defining contracts in Haskell. Contracts are mainly evaluated in an eager manner. Nevertheless, it would be interesting to use their ideas to develop a set of more expressive contract combinators for our framework.

Findler and Felleisen [10] defined a contract system for higher-order functions. In particular, they tackled the problem of correct blame assignment, i.e., to provide precise information about the source position of violated contracts. Although this is orthogonal to the problems addressed in this paper, a correct blame assignment is also relevant in our context and an interesting topic for future work.

Assertions have been also considered in (constraint) logic programming. For instance, [23] proposes a rich assertion language which also includes type and mode information for predicates. [20] combines assertion checking with compile-time verification so that only assertions which cannot be statically verified are dynamically checked. Due to the eager evaluation strategy of Prolog, the difficulties which we address in this paper do not occur there. Nevertheless, it would be interesting to combine our framework with compile-time verification methods.

The demand-driven unification of arguments in our implementation of lazy assertions seems similar to function patterns introduced to obtain more expressive patterns in functional logic programs [3]. In contrast to function patterns, which are completely evaluated to data terms for successful pattern matching, the demand-driven unification introduced in this paper does not evaluate the expressions completely but is driven by the demand of the evaluation context.

5 Conclusions

We have presented a framework to add lazy and enforceable assertions to functional logic programs. Since it is not obvious for any program which properties assertions should satisfy, i.e., whether they should be meaning preserving or faithful, we propose to have both possibilities available in a non-strict language. However, in a functional *logic* language with a demand-driven evaluation strategy, it is reasonable to delay even enforceable assertions as long as possible in order to avoid an unintended exploration of useless branches of the computation space. We have shown an implementation of this framework in the functional logic language Curry, where we used a few non-declarative constructs to implement the assertion framework as a library without modifying the compiler or the run-time system. It should be noted that assertions cause some overhead only if they occur in a program. Since we have not modified the run-time system

to implement assertions, programs without assertions do not have any overhead due to the availability of assertion checking.

For future work, it would be interesting to consider a more expressive assertion language, like the contract language of [18] or the pattern logic of [7]. Furthermore, more work has to be done in order to get a practical tool that automatically provides the source code positions of violated assertions.

Acknowledgements. The author is grateful to the participants of WFLP 2010 and the anonymous referees for helpful comments and suggestions to improve this paper.

References

1. S. Antoy. Optimal Non-Deterministic Functional Logic Computations. In *Proc. International Conference on Algebraic and Logic Programming (ALP'97)*, pp. 16–30. Springer LNCS 1298, 1997.
2. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, Vol. 47, No. 4, pp. 776–822, 2000.
3. S. Antoy and M. Hanus. Declarative Programming with Function Patterns. In *Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05)*, pp. 6–22. Springer LNCS 3901, 2005.
4. S. Antoy and M. Hanus. Functional Logic Programming. *Communications of the ACM*, Vol. 53, No. 4, pp. 74–85, 2010.
5. B. Braßel, O. Chitil, M. Hanus, and F. Huch. Observing Functional Logic Computations. In *Proc. of the Sixth International Symposium on Practical Aspects of Declarative Languages (PADL'04)*, pp. 193–208. Springer LNCS 3057, 2004.
6. O. Chitil and F. Huch. Monadic, Prompt Lazy Assertions in Haskell. In *Proc. APLAS 2007*, pp. 38–53. Springer LNCS 4807, 2007.
7. O. Chitil and F. Huch. A Pattern Logic for Prompt Lazy Assertions in Haskell. In *Proc. of the 18th International Symposium on Application and Implementation of Functional Languages (IFL 2006)*, pp. 126–144. Springer LNCS 4449, 2007.
8. O. Chitil, D. McNeill, and C. Runciman. Lazy Assertions. In *Proceedings of the 15th International Workshop on Implementation of Functional Languages (IFL 2003)*, pp. 1–19. Springer LNCS 3145, 2004.
9. M. Degen, P. Thiemann, and S. Wehr. True Lies: Lazy Contracts for Lazy Languages (Faithfulness is Better than Laziness). In *4. Arbeitstagung Programmiersprachen (ATPS'09)*, pp. 370; 2946–59. Springer LNI 154, 2009.
10. R.B. Findler and M. Felleisen. Contracts for Higher-Order Functions. In *Proceedings of the 7th ACM SIGPLAN international conference on Functional programming (ICFP'02)*, pp. 48–59. ACM Press, 2002.
11. A. Gill. Debugging Haskell by Observing Intermediate Data Structures. *Electr. Notes Theor. Comput. Sci.*, Vol. 41, No. 1, 2000.
12. M. Hanus. A Unified Computation Model for Functional and Logic Programming. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pp. 80–93, 1997.
13. M. Hanus. A Functional Logic Programming Approach to Graphical User Interfaces. In *International Workshop on Practical Aspects of Declarative Languages (PADL'00)*, pp. 47–62. Springer LNCS 1753, 2000.

14. M. Hanus. High-Level Server Side Web Scripting in Curry. In *Proc. of the Third International Symposium on Practical Aspects of Declarative Languages (PADL'01)*, pp. 76–92. Springer LNCS 1990, 2001.
15. M. Hanus. Multi-paradigm Declarative Languages. In *Proceedings of the International Conference on Logic Programming (ICLP 2007)*, pp. 45–75. Springer LNCS 4670, 2007.
16. M. Hanus, S. Antoy, B. Braßel, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/~pakcs/`, 2010.
17. M. Hanus (ed.). Curry: An Integrated Functional Logic Language (Vers. 0.8.2). Available at `http://www.curry-language.org`, 2006.
18. R. Hinze, J. Jeuring, and A. Löh. Typed Contracts for Functional Programming. In *Proc. Eight International Symposium on Functional and Logic Programming (FLOPS 2006)*, pp. 208–225. Springer LNCS 3945, 2006.
19. F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of RTA'99*, pp. 244–247. Springer LNCS 1631, 1999.
20. E. Mera, P. López-García, and M. Hermenegildo. Integrating Software Testing and Run-Time Checking in an Assertion Verification Framework. In *25th International Conference on Logic Programming (ICLP 2009)*, pp. 281–295. Springer LNCS 5649, 2009.
21. B. Meyer. *Object-oriented Software Construction*. Prentice Hall, second edition, 1997.
22. S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.
23. G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Constraint Logic Programs. In *Analysis and Visualization Tools for Constraint Programming*, pp. 23–62. Springer LNCS 1870, 2000.