Version with Proof Appendix and Minor Corrections

Practical Typed Lazy Contracts

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

Abstract

Until now there has been no support for specifying and enforcing contracts within a lazy functional program. That is a shame, because contracts consist of pre- and post-conditions for functions that go beyond the standard static types. This paper presents the design and implementation of a small, easy-to-use, purely functional contract library for Haskell, which, when a contract is violated, also provides more useful information than the classical blaming of one contract partner. From now on lazy functional languages can profit from the assurances in the development of correct programs that contracts provide.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming

General Terms Languages, Reliability

Keywords purely functional, lazy, library, Haskell

1. Introduction

Pre- and post-conditions have been important tools for developing correct programs since the early days of programming. A contract for a function comprises both a pre- and a post-condition. Figure 1 shows definitions in Haskell of two functions with contracts that operate on the type `Formula`, which represents propositional logic formulae. The two functions `clausalNF'` and `clause'` have rather non-descriptive types. The function `clausalNF'` transforms a propositional formula into clausal normal form. To work correctly, the function requires its input to be in conjunctive normal form and to be "right-bracketed", that is, for example `And (Atom 'a') (And (Atom 'b') (Atom 'c'))` is used instead of `And (And (Atom 'a') (Atom 'b')) (Atom 'c')`. The output is a list of list of literals, where a literal is an atom or a negated atom. This pre-condition and post-condition are expressed in the contract `conjNF & right >-> list (list lit)`, which is attached to `clausalNF'` using the function `assert` in the definition of the contracted function variant `clausalNF`. The function `clause` has a similar contract. For any contract `c` the function `assert c` is roughly the identity function, except that it also enforces the contract. The program states the contracts and monitors them at runtime.

Since the work of Findler and Felleisen [12] on contracts for eager functional languages, contracts have become an important item

```
data Formula =
  Imp Formula Formula | And Formula Formula |
  Or Formula Formula | Not Formula | Atom Char

clausalNF =
  assert (conjNF&right >-> list (list lit))
    clausalNF'

clausalNF' :: Formula -> [[Formula]]
clausalNF' (And f1 f2) = clause f1 : clausalNF' f2
clausalNF' f           = [clause f]

clause = assert (disj&right >-> list lit) clause'

clause' :: Formula -> [Formula]
clause' (Or f1 f2) = f1 : clause' f2
clause' lit        = [lit]
```

Figure 1. Contracts of functions for clausal normal form

in the toolbox of the Racket/Scheme programmer. Other functional languages, however, have not yet profited from the support of contracts for several reasons:

- Eager functional contracts were introduced as a small library of contract combinators. However, the implementation in Racket uses its powerful macro system to smoothly integrate contracts into the language[1]. Thus contracts are very easy to use, for example, do not require user-supplied program location parameters. Implementors of other programming languages, however, do not have such a powerful macro system and are wary of making the implementation effort and of extending the language.
- In contrast to dynamically typed Racket, many functional programming languages have a static type system based on Hindley-Milner types with parametric polymorphism. Thus contract combinators need to be statically typed too and it is desirable to have type-directed contract combinators such as `list :: Contract a -> Contract [a]`.

  We also have to avoid classes in types. To see why, consider the following example usage of the Haskell object observation debugger HOOD [16]:

  ```
  length :: Observable a => [a] -> Int
  length = observe "fun" length'

  length' :: [a] -> Int
  length' = List.length
  ```

[1] Recently added features concerning mutable data also required modifications of the language implementation [10].

Here `observe "fun"` behaves like an identity function but also records input and output of the `length'` function for debugging purposes. However, the observation function does not have the type of the identity function: the type of `length` includes the class `Observable` and thus adding an observation may require substantial changes to type annotations in the whole program. To avoid this problem, we have to ensure that contract combinators have simple parametrically polymorphic types, without class contexts.

- Eager functional contracts are strict. Let `nat` be the contract that holds only for non-negative integers. With strict contracts we would get

  ```
  assert (list nat) [4,-1,2] = error "..."
  ```

  Asserting an eager contract yields either the unchanged argument or an error/exception. In contrast, lazy functional languages demand lazy contracts. Asserting a lazy contract yields those parts of the argument data structure that meet the contract; only those parts that violate the contract are "cut off":

  ```
  assert (list nat) [4,-1,2] = [4,error "...",2]
  ```

  Because lazy evaluation generally only evaluates parts of a data structure, a computation may succeed without any contract violation error, if it only demands those data structure parts that meet the contract. Such lazy contracts preserve the lazy semantics of the program and thus ensure that we can add contracts anywhere in a program without changing its semantics, provided contracts are not violated. For example, the following definition of the infinite list of fibonacci numbers requires lazy contracts:

  ```
  fibs :: [Integer]
  fibs = assert (list nat)
           (0 : 1 : zipWith (+) fibs (tail fibs))
  ```

In this paper we develop a library for contracts in Haskell that makes the following contributions:

- The contract combinators have simple parametrically polymorphic types, such that adding a contract does not change the type of a function (Section 2).
- The library provides lazy contract combinators. Adding contracts leaves the semantics of a program unchanged, unless a contract is violated (Sections 2 and 3).
- The library is written in pure, portable Haskell, without any use of side-effecting primitives such as `unsafePerformIO` that could change the semantics of the program (Section 3).
- All data-type-dependent code is simple and thus easy to write by hand if necessary (Section 3).
- The contract combinators have a nice algebra of properties. Contract assertions are partial identities and we claim that they are idempotent too. Thus contracts are projections, like eager contracts (Section 4).
- If a contract is violated, then the raised exception does not simply blame the server (contracted expression) or its client, but provides additional information about the specific value that caused the violation (Section 5).
- The library can use Template Haskell to derive all data-type-dependent code and include source code locations. Thus the programmer can formulate contracts for new algebraic data types without any additional work (Section 6).

The contract library for Haskell is available on Hackage[2].

2. Simple Contract Combinators with a Problem

Using previous work on eager contracts [11] and typed contracts [19], we can easily design and implement most of a contract library for Haskell.

We implement a parametric type `Contract a` and a function

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

that turns a contract into a partial identity, that is, `assert c` $\sqsubseteq$ `id`. Here $\sqsubseteq$ is the standard information-theoretic partial order on values with least element $\bot$. For simplicity we consider $\bot$ to be an expression. It represents both non-termination and an exception raised by a violated contract.

Most of the contract library consists of combinators for building contracts of type `Contract T`, for various types `T`.

We start with a combinator that turns a predicate into a contract:

```
prop :: Flat a => (a -> Bool) -> Contract a
```

used for example as

```
nat :: Contract Integer
nat = prop (>=0)
```

to specify natural numbers as integers greater or equal zero.

We have to restrict `prop` by a new class `Flat` to be used for flat types only. A type is flat if for all values v_1 and v_2 the ordering $v_1 \sqsubset v_2$ implies $v_1 = \bot$. We cannot use `prop` for non-flat types such as lists, because the predicate could be arbitrarily strict and thus violate our aim of building lazy contracts [2]. For example

```
nats' = prop (all (>=0)) :: Contract [Integer]
```

would not be lazy and thus would be unusable for our infinite list of fibonacci numbers.

So the class `Flat` has only a few instances such as

```
instance Flat Integer
instance Flat Float
instance Flat Char
```

In our initial example in Figure 1 we already used two combinators for building contracts:

```
(&)   :: Contract a -> Contract a -> Contract a
(>->) :: Contract a -> Contract b -> Contract (a -> b)
```

The conjunction combinator `(&)` builds a contract that is violated if one of the components is violated. The combinator `(>->)` looks similar to the function type; it does *not* indicate logical implication. The function combinator combines a pre- and a post-condition to a contract for a function. For a function to be be correct, whenever the pre-condition holds, the post-condition must hold too. However, if the pre-condition is violated, then the client (caller) of the function is wrong. A function contract is an agreement between both a function and its client. So neither pre- nor post-condition should be violated. In summary, the function contract combinator is rather like the conjunction combinator, except that values of two possibly different types are monitored.

A contract that is always met is useful as component of a bigger contract to express that some values are irrelevant. The opposite contract that is never met can still be occasionally useful in a lazy functional language. We can use

```
true  :: Contract a
false :: Contract a
```

for example in

[2] `http://hackage.haskell.org`

```
type Contract a = a -> a

assert c = c

class Flat a where
  prop :: (a -> Bool) -> Contract a
  prop p = \x -> if p x then x else error "..."

pNil []    = []
pNil (_:_) = error "..."

pCons c cs [] = error "..."
pCons c cs (x:xs) = c x : cs xs

true  = id
false = const (error "...")

c1 & c2      = c2 . c1
pre >-> post = \f -> post . f . pre
```

Figure 2. A lazy contract implementation for most combinators

```
const = assert (true >-> false >-> true) const'

const' :: a -> b -> a
const' x y = x
```

to express that the second argument of the function is never demanded. Because that argument is never demanded, its contract will never be used and thus will never be violated.

Finally we need combinators to build contracts for algebraic data types, which generally are not flat. Here we introduce for each data constructor a combinator that is used like a data constructor in pattern matching.

```
pNil  :: Contract [a]
pCons :: Contract a -> Contract [a] -> Contract [a]
```

Now we can define a contract for infinite lists:

```
infinite :: Contract [a]
infinite = pCons true infinite
```

If this contract is asserted for a finite list and evaluation demands the last constructor, `[]`, of this finite list, then a contract violation exception is raised. Here we also use the contract `true` to state that we do not restrict the list elements in any way.

All our combinators can be implemented using the same contract type as for the well-known eager contracts. Even the new data constructor combinators can easily be implemented using that type. The short implementation is given in Figure 2.

However, one important combinator is still missing. On their own, data constructor combinators such as `pNil` and `pCons` are of rather limited use, the infinite list contract being one of the few examples where they suffice. We need a combinator for combining two data constructor contracts disjunctively:

```
(|>) :: Contract a -> Contract a -> Contract a
```

This combinator allows us, for example, to define the contract of a (finite or infinite) list of natural numbers as follows:

```
nats :: Contract [Integer]
nats = pNil |> pCons nat nats
```

The definition intentionally looks very similar to the definition of an algebraic data type.

We cannot define `(|>)` using the contract type definition

```
type Contract a = a -> a
```

```
type Contract a = a -> Maybe a

assert :: Contract a -> (a -> a)
assert c x = case c x of
               Just y  -> y
               Nothing -> error "Contract violated."

class Flat a where
  prop :: (a -> Bool) -> Contract a
  prop p x = if p x then Just x else Nothing

pNil :: Contract [a]
pNil []    = Just []
pNil (_:_) = Nothing

pCons :: Contract a -> Contract [a] -> Contract [a]
pCons c cs []     = Nothing
pCons c cs (x:xs) = Just (assert c x : assert cs xs)

true :: Contract a
true = Just

false :: Contract a
false = const Nothing

(|>) :: Contract a -> Contract a -> Contract a
c1 |> c2 = \x -> c1 x `mplus` c2 x

(&) :: Contract a -> Contract a -> Contract a
c1 & c2 = \x -> c1 x >>= c2

(>->) :: Contract a -> Contract b -> Contract (a -> b)
pre >-> post =
  \f -> Just (f `seq` (assert post . f . assert pre))
```

Figure 3. Implementation of typed lazy contract combinators

We can combine two functions of type `a -> a` only by composition and we have done so already for the contract combinator `(&)`. For disjunction we would need to apply both functions separately and then somehow combine the two results: if one is an exception, then we should return the value of the other one. We cannot test for exceptions in a purely functional language[3].

3. Implementing Lazy Contract Combinators

A simple modification of our contract type definition solves our problem:

```
type Contract a = a -> Maybe a
```

The `Maybe a` type enables us to test for contract violation and then to try the next contract. The return value `Nothing` indicates that the contract is violated for the top constructor of the monitored value. The return value `Just v` indicates successful matching of the top constructor and returns the value with possibly further contracts attached to its components.

Recall that for `Maybe a` its monadic functions are defined as follows:

```
(>>=) :: Maybe a -> (a -> Maybe b)
(Just x) >>= f = f x
Nothing  >>= f = Nothing
```

[3] There is an impure solution [6] that, however, still cannot handle non-termination in one argument.

```
mplus :: Maybe a -> Maybe a -> Maybe a
(Just x) `mplus` m = Just x
Nothing  `mplus` m = m
```

Figure 3 lists the full implementation of our contract combinators.

In the definition of the function contract combinator `seq` first evaluates the function itself before returning it wrapped in assertions. This definition ensures

$$\texttt{assert (}c_1\texttt{ >-> }c_2\texttt{) } \bot = \bot$$

that is, function contracts are strict like all other contracts. Without `seq` the expression `assert (`c_1` >-> `c_2`) `$\bot$ instead would be the function that demands no argument and always returns $\bot$, which in Haskell can be distinguished from the function $\bot$ itself. Thus the function contract combinator would change the semantics of a program even when the contract is not violated. Admittedly, we do not expect this case to ever occur in practice.

The contract type reminds of parser combinators. However, contracts are deterministic: contract application `assert c` is a function for any contract `c`. No value of a second type, e.g. a parse tree, is constructed. Hence we only need the `Maybe a` type with its two choices, not the more general list type `[a]` that would provide an arbitrary number of choices.

A pattern combinator tests only for the top constructor. If that fits, then the pattern combinator succeeds. Hence

```
assert (pCons nat pNil |> pCons true pNil) [-3] =
   [error "Contract violated."]
```

Here matching the list constructor of `pCons nat pNil` succeeds and therefore the second list contract `pCons true pNil` is never tried, even though the contract for the list element, `nat`, is violated. For all our examples the simple semantics suffices and it is easy to understand.

For example, we can define a parameterised contract for the list data type

```
list :: Contract a -> Contract [a]
list c = pNil |> pCons c (list c)
```

and use it to define the contract of list of natural numbers:

```
nats :: Contract [Integer]
nats = list nat
```

We can also define functions with non-contract parameters to construct contracts:

```
listOfLength :: Int -> Contract [a]
listOfLength 0 = pNil
listOfLength (n+1) = pCons true (listOfLength n)
```

However, in such a case we need to be sure that the parameter value is well-defined, so that it cannot introduce non-termination into the program. For example

```
lengthAtLeast :: Int -> Contract [a]
lengthAtLeast 0    = true
lengthAtLeast (n+1) = pCons true (lengthAtLeast n)

contractTake :: Int -> [a] -> [a]
contractTake n =
  assert (lengthAtLeast n >-> listOfLength n)
    (take n)
```

is only safe, because the function `take` is strict in its integer parameter, which determines how many list elements shall be returned.

$$\texttt{prop } p_1 \texttt{ |> prop } p_2 = \texttt{prop (\textbackslash x -> } p_1\texttt{ x || } p_2\texttt{ x)}$$
$$\texttt{prop } p_1 \texttt{ \& prop } p_2 = \texttt{prop (\textbackslash x -> } p_1\texttt{ x \&\& } p_2\texttt{ x)}$$

$$c_1 \texttt{ \& (}c_2\texttt{ \& }c_3\texttt{)} = \texttt{(}c_1\texttt{ \& }c_2\texttt{) \& }c_3$$
$$\texttt{true \& c} = \texttt{c}$$
$$\texttt{c \& true} = \texttt{c}$$
$$\texttt{false \& c} = \texttt{false}$$

$$c_1 \texttt{ |> (}c_2\texttt{ |> }c_3\texttt{)} = \texttt{(}c_1\texttt{ |> }c_2\texttt{) |> }c_3$$
$$\texttt{false |> c} = \texttt{c}$$
$$\texttt{c |> false} = \texttt{c}$$
$$\texttt{true |> c} = \texttt{true}$$
$$\texttt{c |> c} = \texttt{c}$$
$$c_1 \texttt{ |> (}c_2\texttt{ |> }c_1\texttt{)} = c_1 \texttt{ |> } c_2$$

$$c_1 \texttt{ |> (}c_1\texttt{ \& }c_2\texttt{)} = c_1$$

$$\texttt{true >-> true} = \texttt{true}$$
$$c_1 \texttt{ >-> false} = c_2 \texttt{ >-> false}$$
$$\texttt{(}c_1\texttt{ >-> }c_2\texttt{) \& (}c_3\texttt{ >-> }c_4\texttt{)} = \texttt{(}c_3\texttt{ \& }c_1\texttt{) >-> (}c_2\texttt{ \& }c_4\texttt{)}$$
$$\texttt{(}c_1\texttt{ >-> }c_2\texttt{) |> (}c_3\texttt{ >-> }c_4\texttt{)} = c_1 \texttt{ >-> } c_2$$

Figure 4. Contract properties

$$c_1 \texttt{ \& (}c_1\texttt{ |> }c_2\texttt{)} = c_1$$
$$c_1 \texttt{ \& (}c_2\texttt{ \& }c_1\texttt{)} = c_1 \texttt{ \& } c_2$$
$$\texttt{c \& c} = \texttt{c}$$

Figure 5. Claimed contract properties

4. Properties of Contracts

Our contract type `Contract a` is a combination of the function type with the `Maybe` monad. Thus we have a rich set of known properties to work with for establishing an algebra of contracts. `Contract` itself is *not* a monad.

4.1 An Algebra of Contracts

Figure 4 lists many simple properties enjoyed by contracts. All of these can be proved by simple equational reasoning, using the monad laws of `Maybe a`.

All the properties of conjunction and disjunction of contracts also hold for conjunction and disjunction of Booleans in a lazy language. Recall that some standard properties of Boolean algebra do not hold for the Boolean type in non-strict languages. For example, `(&&)` and `||` are not commutative and the standard distribution laws do not hold. These properties do not hold for contracts either, with similar counterexamples. So the non-strict algebra of `(&&)` and `||` is a good guideline for developing the lazy contract algebra of `&` and `|>`.

The ”distribution” law for conjunction and function contract may at first surprise. It holds because the function contract combinator is not some kind of implication but more a kind of conjunction. A function contract holds only if both the input and the

output of a function meet the respective subcontracts. From this "distribution" law of conjunction and function plus idempotence of conjunction further laws follow:

$$(c_1 \texttt{ >-> } c) \texttt{ \& } (c_3 \texttt{ >-> } c) = (c_1 \texttt{ \& } c_3) \texttt{ >-> } c$$

$$(c \texttt{ >-> } c_2) \texttt{ \& } (c \texttt{ >-> } c_4) = c \texttt{ >-> } (c_2 \texttt{ \& } c_4)$$

Figure 5 lists further properties of contracts that we have not proved but claim also hold. They require stronger proof methods than equational reasoning, but are linked to the idempotence of contracts discussed in the subsequent subsection. The last property in the list, idempotence of conjunction, is a corollary of the preceeding property, taking $c_2 = \texttt{true}$.

4.2 Contracts are Projections

Eager contracts are projections [1, 11], that is, they are idempotent and partial identities.

Lemma 4.1 (A contract is a partial identity)**.**
For any contract `c`

$$\texttt{assert c} \sqsubseteq \texttt{id}$$

This can be proved using induction on the contract combinators.

Idempotence is more difficult to establish. It would follow from idempotence of conjunction, `c & c` $=$ `c`. In practice, both properties probably need to be established in a single inductive proof. Intuitively idempotence holds because if a contract returns `Just v`, that value `v` is the same as would be returned by the eager contract type `a -> a`, and pattern contracts only test for the top constructor before returning `Just v` or `Nothing`.

Claim 4.2 (A contract is idempotent)**.**
For any contract `c`

$$\texttt{assert c . assert c} = \texttt{assert c}$$

4.3 Distinct Contract Exceptions

We identified non-termination and any contract exception as the single value $\bot$. However, we might distinguish them, following [22], such that exceptions are values above $\bot$ in the information order, but still values of any type. We would change our partial order to consider exceptions as least elements, because a contract replaces some parts of values by exceptions. However, with that choice our contracts are neither partial identities, nor idempotent (the properties of Figure 4 are unaffected). The reason is that contracts such as

```
⊥ :: Contract a
prop ⊥ :: Flat a => Contract a
prop (\x -> if odd x then True else ⊥) :: Contract Int
```

exist. They would still introduce $\bot$ instead of exceptions.

For related reasons other works [1, 8] restricted the definitions of contracts such that a contract can never introduce $\bot$ itself. However, the desirable freedom to use the whole language to define contracts and the fact that we are just defining a library makes this an impractical choice.

5. Informative Contract Violation

A contract is concluded between two partners, a server and a client. If a contract is violated, one of the two partners is to blame for it. A major contribution of Findler and Felleisen's functional contracts [12] is its system for choosing whom to blame. In a higher-order language function arguments can themselves be functions. If such a functional argument is used within the function such that the precondition of the functional argument is violated, then the function itself has to be blamed for contract violation, not the caller that passed the functional argument.

```
type Contract a = a -> Bool -> Either Bool a

assert :: Contract a -> (a -> a)
assert = monitor True

monitor :: Bool -> Contract a -> (a -> a)
monitor b c x =
  case c x b of
    Right x -> x
    Left b  -> error ("Contract violated. Blame "
                      ++ if b then "server."
                              else "client.")

(>->) :: Contract a -> Contract b -> Contract (a->b)
pre >-> post = \f b -> Right (f `seq`
  (monitor b post . f . monitor (not b) pre))

true :: Contract a
true = \x b -> Right x

false :: Contract a
false = \x b -> Left (not b)
```

Figure 6. Implementing blaming

5.1 Blaming

The blaming system for higher-order functional languages applies to both eager and lazy languages equally, and thus we can easily add it to our lazy contract library. For eager languages several equivalent implementations for handling blame are known [11, 12, 19]. Here we simply extend a contract by a Boolean state that indicates whether the server or the client of the contract are to blame in case of violation. The `Maybe` monad is replaced by `Either Bool a` so that blame information is available when a subcontract is violated. Figure 6 shows the most interesting extended definitions. Contract monitoring starts by potentially blaming the server, that is, the expression for which the contract is asserted. The function contract combinator `>->` negates the Boolean blame indicator for monitoring the contra-variant argument, but passes it unchanged for monitoring the co-variant result.

Now there are two different possible implementations of the contract `false` that can never be met: The contract either always blames the party indicated by the given Boolean argument, or it always blames the opposite party by negating the Boolean value. So let us look back at our example of Section 2:

```
const = assert (true >-> false >-> true) const'
```

Any client of `const` will provide some second argument, but if that second argument is actually demanded, then clearly `const'` is wrongly defined and has to be blamed. In this example `false` is in a contra-variant position of the whole contract and hence to blame the server, `false` has to negate its Boolean parameter. So on its own, `false` always blames its client, never its server. We do not provide the server-blaming variant in the library, because it does not seem to be of any practical use.

5.2 Witness Tracing

Blaming alone, however, is rather unsatisfactory. It just points the finger at one partner without providing any evidence that would explain in which way a complex contract was violated. Blaming hardly provides a good starting point for debugging. Furthermore, blaming can be misleading. Often when a contract is violated neither server nor client are wrong, but the contract itself! Specifying

```
type Contract a =
  (String->String) -> a -> Either String a

assert :: Contract a -> (a -> a)
assert = monitor id

monitor :: (String->String) -> Contract a -> (a->a)
monitor wc c x =
  case c wc x of
    Right v -> v
    Left w  -> error ("Contract violated. Witness:"
                      ++ wc ("{" ++ w ++ "}"))

(>->) :: Contract a -> Contract b -> Contract (a->b)
pre >-> post = \wc f -> Right (f `seq`
  (monitor (wc . \w->("(_->"++w++")")) post . f .
   monitor (wc . \w->("("++w++"->_)")) pre))

pNil :: Contract [a]
pNil = \wc x -> case x of
  []  -> Right x
  _:_ -> Left "_:_"

pCons :: Contract a -> Contract [a] -> Contract [a]
pCons cx cxs = \wc x -> case x of
  (y:ys) ->
    Right
      (monitor (wc . \w->("("++w++":_)")) cx y :
       monitor (wc . \w->("(_:"++w++")")) cxs ys)
  []     -> Left "[]"
```

Figure 7. Implementing witness tracing

the right contract is challenging and contract monitoring just checks whether specification and implementation agree.

Hence our lazy contracts report, when they are violated, the top data constructor, or whole flat value, that causes the contract violation, plus all data constructors in the path above it. For example

```
*Main> clausalNF form
[[Atom 'a'],[Atom 'b',Not
*** Exception: Contract violated. Witness:
((And _ (Or _ (Not {Not _})))->_)
```

Here we do not need to know the full definition of the formula `form`. The error message tells us all that we need to know: The formula contains a double-negation and therefore is not in conjunctive normal form, as the contract of `clausalNF` requires. More precisely, the contract was asserted for a function that took as argument a formula with `And` at the top, with `Or` as second argument, which has a `Not` as second argument, which has the forbidden `Not` as argument.

To trace the required information of a potential witness of contract violation, our contracts pass an additional argument that accumulates a description of the context of a monitored value, and a violated contract returns a string describing the offending value itself. The representation of the context is of type `String -> String` to easily slot another context or expression representation into the hole of the context. Figure 7 gives an outline of the implementation. The printed witness describes just the data that needs to be evaluated to notice the contract violation.

5.3 Location + Blame + Witness

Our final contract library combines blaming and witness tracing, records the source location of a contract and raises a special exception to provide the maximal information when the contract is violated. For example:

```
*Main> clausalNF form
[[Atom 'a'],[Atom 'b',Not
*** Exception: Contract at ContractTest.hs:101:3
violated by
((And _ (Or _ (Not {Not _})))->_)
The client is to blame.
```

6. Deriving Contract Combinators

For every data constructor `Con` that we want to pattern match in a contract we have to define a pattern contract `pCon`. These definitions are simple, even with handling of location, blame and witness information, but they are still tedious. Hence our contract library allows their automatic derivation using Template Haskell [23]. Template Haskell is a meta-programming extension of Haskell that the Glasgow Haskell compiler, the only Haskell system used for professional Haskell program development, provides. Template Haskell allows us to define in the contract library functions that will generate Haskell code at compile time, type check that code and compile it.

The user no longer needs to define these pattern contracts at all, but can basically derive them on demand where needed, that is, directly write

```
conjNF = $(p 'And) conjNF conjNF |> disj
disj   = $(p 'Or)  disj disj |> lit
lit    = $(p 'Not) atom |> atom
atom   = $(p 'Atom) true
```

Here `p` is a Template Haskell function that receives the name of a data constructor as argument. The `$` and the single quote in front of the data constructor are syntax required by Template Haskell. The definition of a pattern contract is short so that repeated derivation is not a problem.

Alternatively, the programmer can also write the declaration

```
$(deriveContracts ''Formula)
```

to derive all pattern contract definitions for the type `Formula`.

Finally, `assert` is also a Template Haskell function:

```
clausalNF =
  $assert (conjNF & right >-> list (list lit))
    clausalNF'
```

Template Haskell allows the definition of `assert` to determine its own location in the file and then generate code for calling the real assertion function with that location as parameter.

7. Further Contract Features

Initial experience of using contracts raises new questions and demand for additional contract combinators.

7.1 Negation

We have conjunction, `&`, and disjunction, `|>`, of contracts. However, we cannot have negation

```
neg :: Contract a -> Contract a
```

for contracts. General negation would violate basic semantic properties of contracts [2].

Nonetheless, in practice we often want to express that the top data constructor of a monitored value is *not* a specific given data constructor. Hence we introduce additional combinators such as the following for every data type.

```
pNotImp :: Contract Formula
```

```
pNotAnd :: Contract Formula
pNotOr :: Contract Formula
pNotNot :: Contract Formula
pNotAtom :: Contract Formula
```

These negated pattern contracts provide nothing new. In fact

```
pNotImp = pAnd true true |> pOr true true |>
          pNot true |> pAtom true
```

However, for types with many data constructors these combinators are certainly substantial abbreviations and they are needed frequently. Additionally, our implementation can perform an efficient single pattern match instead of many repeated ones.

We use these negated pattern contracts in the definition of contracts for our initial propositional formulae example. They substantially simplify our definition of "right-bracketedness".

```
conjNF, disj, lit, atom,
  right, rightConjNF :: Contract Formula

conjNF = pAnd conjNF conjNF |> disj
disj   = pOr disj disj |> lit
lit    = pNot atom |> atom
atom   = pAtom true

right = pImp (right & pNotImp) right |>
        pAnd (right & pNotAnd) right |>
        pOr (right & pNotOr) right |>
        pNot right |> pAtom true

rightConjNF = conjNF & right
```

Even for data types with few constructors they can express an idea more clearly. So

```
head' = assert (pNotNil >-> true) head
```

is more direct than

```
head' = assert (pCons true true >-> true) head
```

to express that the function only works on non-empty lists.

7.2 Contracts for the IO monad

We have contract combinators for flat types, algebraic data types and the function type constructor. However, a real programming language has more types, especially abstract data types. The most notorious in Haskell is the IO monad that is required for any input or output actions.

For example, we may want to write a contract for an IO action that gets a natural number from standard input:

```
getNat :: IO Integer
getNat = assert (io nat) getNat'
```

The choice of contract combinator is natural, following our general approach of type-directed contract combinators. How do we define the IO contract combinator?

```
io :: Contract a -> Contract (IO a)
io c = \io -> Just (io >>= return . assert c)
```

Our definition simply follows the scheme we are already using for the function contract combinator `>->`. After all, the function type is "just" an abstract data type as well[4]. With this definition our IO contract combinator also has the same properties as the function contract combinator:

$$\texttt{io } c_1 \texttt{ \& io } c_2 = \texttt{io } (c_1 \texttt{ \& } c_2)$$

$$\texttt{io } c_1 \texttt{ |> io } c_2 = \texttt{io } c_1$$

$$\texttt{io true} = \texttt{true}$$

Our definition of the contract combinator for the abstract data type `IO a` raises the question whether we should do the same for other data types. For example, we have

```
list :: Contract a -> Contract [a]
list c = pNil |> pCons c (list c)
```

Alternatively we could follow our definition of `io`:

```
list' c = \xs -> Just (xs >>= return . assert c)
```

which is the same as

```
list' c = \xs -> Just (map (assert c) xs)
```

It turns out that the two definitions are equivalent[5], thus confirming our original definition. Hence we prefer to define a contract combinator for a non-abstract algebraic data type such as `list` in terms of the only primitive contract combinators, the pattern contract combinators, such as `pNil` and `pCons`. For an abstract data type we define a contract combinator using the scheme above with the respective map function for the type.

7.3 Strict data types

Haskell allows the definition of strict data types. The strictness flag `!` in a data type definition states that the data constructor is strict in that argument. For example,

```
data SListBool = SNil | SCons !Bool !SListBool
```

defines the type of finite Boolean lists that are either $\bot$ or fully defined.

Happily we do not need to adapt our definition of contract combinators. As usual we have

```
pSNil :: Contract SListBool
pSNil SNil = Just SNil
pSNil (SCons b bs) = Nothing

pSCons :: Contract Bool -> Contract SListBool ->
          Contract SListBool
pSCons c cs SNil = Nothing
pSCons c cs (SCons b bs) =
  Just (SCons (assert c b) (assert cs bs))
```

A contract traverses the strict list and builds a new strict list. Thus demanding the top data constructor of a contracted strict list automatically forces checking the whole list. The result will be either a contract violation ($\bot$) or the whole list. So on strict data types lazy contracts behave like eager contracts. It would be possible to define more expressive contract combinators for data types with strictness flags, that, for example, ensure that a list is ordered; but because strictness flags are rarely used in Haskell programs, such an extension does not seem worthwhile.

In the definition of `SListBool` all constructor arguments are strict and the only other types used are flat types. In such a case the data type is actually a flat type. We can declare it an instance of the class `Flat` and use expressive `prop` contracts.

[4] The forced evaluation with `seq` by the function contract combinator is required because of the peculiar semantics of functions in Haskell. It is not needed for other types. For example, if `io` $= \bot$, then also `io >>= return . assert c` $= \bot$.

[5] They are not equal, because `list` $\bot$ $=$ $\bot$, but `list'` $\bot$ `= Just` $\bot$. However, in the context of a contract with `assert` they always yield the same result.

8. Related Work

This paper builds firmly on three sets of previous work: Findler and Felleisen's work on eager contracts for higher-order functions [12], Hinze, Jeuring and Löh's work on typed contracts for functional programming, and our own previous work on lazy functional contracts.

Eager Higher-Order Contracts Findler and Felleisen's paper on contracts for higher-order functions [12] made contracts popular for eager functional programming languages. All interesting properties of functional values, which are passed around by higher-order functions, are undecidable; it is impossible to monitor a function contract for all argument-result-pairs. However, Findler and Felleisen realised that it is sufficient to monitor both pre- and post-condition of a functional value only when this function is applied. The resulting contract system is sound.

The second major contribution of that paper is a system for correctly attributing blame in case of contract violation and its implementation. We easily added blaming to our lazy contract library. However, additionally our contracts report a witness, a partial value, that caused a contract violation.

Findler and Felleisen's contract system also provides dependent function contracts, where the contract for the function codomain can use the actual argument value. Such dependent contracts are more expressive but easily change a non-strict function into a strict function; hence our lazy contract combinators do not provide them.

Subsequent work [11, 13] stresses that contracts are projections and thus they can be implemented in a simple, modular and efficient way. Our first implementation of Figure 2 copies that work and our full implementation with disjunction is an extended variant.

The papers do not discuss algebraic data types, because in strict languages these domains are flat and hence contracts for algebraic data types are predicates like for other flat types. Consequently disjunction is not considered either. Because of the universality of predicate contracts in strict and dynamically typed functional languages, type-directed contracts such as `list` are also of little interest.

Although disjunction is not discussed in the papers, the contract system of Racket does provide a disjunctive contract combinator [14, Version 5.2.1] [15]. To support disjunction, a Racket contract for type `a` contains both a function of type `a -> a` and a function of type `a -> Bool`. Together they are used similarly to our type `a -> Maybe a`. In particular, the disjunctive combinator applies the `a -> Bool` functions of all its direct sub-contracts, checks that at most one of the results is `True` (otherwise it fails) and then applies the corresponding sub-contract further [10]. So disjunction behaves similarly to `|>` but is not sequential.

Blume et al. proposed and studied several semantic models of eager higher-order contracts [1, 11, 13]. To prove soundness, the definition of contracts is first restricted, to avoid e.g. having a contract $\bot$. Later, recursive contracts are added to regain expressivity. In a discussion of the most permissive contract, `true`, Findler and Blume point out that the contract `true`, to be the most permissive contract, should always report contract violation and blame the client. However, they also note that such a contract would be useless in practice. In contrast, our `true`, which cannot be violated, is very useful to leave parts of a contract unconstrained. Similarly, the least permissive contract should always blame the server, but we demonstrated in Section 5.1 that our definition of `false`, which always blames the client, is more useful. As a consequence in our library `false = prop (const False)` does *not* hold for flat types whereas `true = prop (const True)` does.

Typed Contracts for Functional Programming Hinze, Jeuring and Löh [19] transferred contracts for higher-order functions to the statically typed language Haskell. Hence they proposed contract combinators with parametrically polymorphic types; we have adopted all of them except for dependent contracts. Typed contracts also emphasis type-directed contract combinators such as `list`. However, the work disregards the lazy semantics of Haskell, defining contracts with a seemingly random mixture of eager and lazy monitoring. Predicate contracts can be applied to expressions of all types, not just flat types, thus breaking laziness. However, these predicate contracts are required for expressing many interesting properties, because a type-directed contract combinator such as `list` can only express a uniform property over all list elements: our pattern contracts and disjunction are missing.

Contracts are not projections, because generally they are not idempotent. Idempotence is lost because of the eagerness of predicate contracts. Hinze et al. make the point that if contract conjunction `&` was commutative, then idempotence would be a simple consequence. However, our lazy contracts demonstrate that commutativity of conjunction is not necessary for idempotence; we can have the latter without the former.

Hinze et al. also provide an interesting technique for providing more informative error messages than standard blaming. Their library provides several source locations as explanation of a single violated contract. However, these sets of source locations are still hard to understand for a programmer and the system requires a source code transformation to insert source locations into the program. Otherwise the programmer would have to do this substantial work.

Lazy Contracts for Functional Languages Lazy contracts were first discussed and several implementations presented in 2004 [5]. That paper makes the point that while eager contracts must be `True`, lazy contracts must not be `False`. This means that unevaluated parts of a data structure can never violate a lazy contract. The paper uses predicates on values of all types and hence, despite some technical tricks using concurrency, the contracts are lazy but neither idempotent nor prompt. The paper itself gives examples of where contract violations are noticed too late. This problem was later rectified [3, 4]. Both these papers implement lazy assertions as libraries that require only the commonly provided non-pure function `unsafePerformIO`, which performs side effects within a purely functional context. The first lazy and idempotent implementation [4] uses patterns contracts similar to those in this paper to express contracts over algebraic data types. However, a non-deterministic implementation of disjunction leads to semantic problems. Later [3] provided a more user-friendly language for expressing contracts and improved the internal structure of the implementation, but the implementation principles were identical and hence the non-deterministic disjunction remained.

A semantic investigation [2] developed contracts that are pure and implementable within the functional language. However, for every algebraic data type its contracts requires a different implementation type. Thus disjunction is not a parametrically polymorphic combinator but requires a class context. Furthermore, the implementations of some combinators are large and complex. Disjunction is more powerful than in the lazy contracts described in the present paper, for example

```
  assert (pCons nat pNil |> pCons true pNil) [-3]
=assert (pCons (nat |> true) (pNil |> pNil))
=[-3]
```

but this additional expressibility does not seem to be needed in practice.

Comparing Contracts Degen, Thiemann and Wehr [7, 8] classify existing contract systems for Haskell as eager (straight translation of [12]), semi-eager [19] and lazy [3–5]. They check whether the systems meet their desirable properties of meaning preservation

and completeness. Each contract system meets at most one of these properties. The authors show that it is impossible to meet both properties. Our lazy contracts are meaning preserving but not complete. The notion of completeness seems to be biased towards a strict semantics, contradicting the principle that unevaluated parts can never violate a lazy contract. Our lazy contracts have limited expressibility, but they have a clear semantics.

Generic Programming We use Template Haskell to derive pattern contracts and to enable the assertion function to determine its own location in the source code [23]. The derivation of pattern contracts is an instance of generic programming. Many generic programming systems have been proposed and even been implemented for Haskell [17, 18, 20, 21]. All of these have two disadvantages that make them unsuitable for being used for our pattern contracts: First, they introduce one or more classes that will then appear in the type of every derived pattern contract. Thus pattern contracts will not be parametrically polymorphic. Second, they consider functions as second class values. That means that either they can only generically define code for types that do not involve function types at all, or they can recognise a functional value within an algebraic data type, but cannot do anything with it, that is, apply any transformation to it.

Template Haskell provides few static guarantees and thus requires us to ascertain that our contract library will derive typeable and correct code for any data constructor. However, Template Haskell provides all the functionalities needed in the contract library.

9. Conclusions and Future Work

This paper describes the design of a practical contract library for lazy typed functional languages and its implementation for Haskell. The library meets many essential criteria, such as combinators with simple parametric types, a lazy semantics, a rich algebra of properties, informative exceptions in case of contract violation and automatic code generation to make it easy to use.

Interestingly the resulting contract system reminds strongly of a subtyping system, especially with a definition of sub-contracts/types for algebraic data types that looks very similar to the actual type definition of algebraic data types. Defining subtypes of algebraic data types is also where we see the main application area of the contract system. Many programs require several variants of some big algebraic data types. In practice programmers then simply ignore the subtyping and define a single algebraic data type that encompasses all variants, because they want to reuse functions that work on several subtypes and have the flexibility to exchange some code without having to change between numerous similar but separate data types. The classical example is a compiler: it consists of a long sequence of passes, each of which works with a slightly differently structured abstract syntax tree, In practice, subtle differences are ignored and only a few different abstract syntax tree structures are used in one compiler. Lazy contracts provide a new solution.

Our next step is to develop the algebra of contract combinators further and thus also prove our claim that these contracts are idempotent. The main current shortcoming and thus biggest challenge for future development of the contract library is its lack of a dependent function contract combinator that allows using the function argument in the post-condition. We can define

```
(>>->) :: Contract a -> (a -> Contract b) ->
            Contract (a -> b)
pre >>-> post = \f ->
  Just (f `seq` \x -> let y = assert pre x
                      in assert (post y) (f y))
```

and use it for example in

```
contractTake :: Int -> [a] -> [a]
contractTake =
  assert (prop (>=0) >>->
          \n -> lengthAtLeast n >-> listOfLength n)
    take
```

It is easy to extend this picky implementation to use indy monitoring [9], which may blame the contract itself, not just the server or the client. However, `>>->` is not a lazy contract combinator; the post-condition may force evaluation of too much of the function argument and thus the contract may change the semantics of the program. A definition of a lazy dependent function combinator is still an open problem. Meanwhile the existing contract library can be used in practice.

Acknowledgments

I thank Simon Thompson and Stefan Kahrs for useful advice on early versions of this work and the anonymous ICFP reviewers and Robby Findler for their detailed comments.

References

[1] M. Blume and D. McAllester. Sound and complete models of contracts. *J. Funct. Program.*, 16(4-5):375–414, 2006.

[2] O. Chitil. A semantics for lazy assertions. In *Proceedings of the 20th ACM SIGPLAN workshop on Partial evaluation and program manipulation*, PEPM 2011, pages 141–150, January 2011.

[3] O. Chitil and F. Huch. Monadic, prompt lazy assertions in Haskell. In *APLAS 2007*, LNCS 4807, pages 38–53, 2007.

[4] O. Chitil and F. Huch. A pattern logic for prompt lazy assertions in Haskell. In *Implementation and Application of Functional Languages: 18th International Workshop, IFL 2006*, LNCS 4449, 2007.

[5] O. Chitil, D. McNeill, and C. Runciman. Lazy assertions. In *Implementation of Functional Languages: 15th International Workshop, IFL 2003*, LNCS 3145, pages 1–19. Springer, November 2004.

[6] N. A. Danielsson and P. Jansson. Chasing bottoms, a case study in program verification in the presence of partial and infinite values. In D. Kozen, editor, *Proceedings of the 7th International Conference on Mathematics of Program Construction, MPC 2004*, LNCS 3125, pages 85–109. Springer-Verlag, July 2004.

[7] 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)*, Lübeck, Germany, October 2009.

[8] M. Degen, P. Thiemann, and S. Wehr. The interaction of contracts and laziness. In *Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation*, PEPM '12, pages 97–106, 2012.

[9] C. Dimoulas, R. B. Findler, C. Flanagan, and M. Felleisen. Correct blame for contracts: no more scapegoating. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 215–226, 2011.

[10] R. B. Findler. Comparison with Racket's contract system. Personal communication, 2012.

[11] R. B. Findler and M. Blume. Contracts as pairs of projections. In *International Symposium on Functional and Logic Programming (FLOPS)*, LNCS 3945, pages 226–241, 2006.

[12] 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, 2002.

[13] R. B. Findler, M. Blume, and M. Felleisen. An investigation of contracts as projections. Technical report, University of Chicago Computer Science Department, 2004. TR-2004-02.

[14] M. Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Inc., 2010. `http://docs.racket-lang.org/trl/`.

[15] M. Flatt, R. B. Findler, and PLT. The Racket guide. `http://docs.racket-lang.org/guide/index.html`, 2012.

[16] A. Gill. Debugging Haskell by observing intermediate datastructures. *Electronic Notes in Theoretical Computer Science*, 41(1), 2001. (Proc. 2000 ACM SIGPLAN Haskell Workshop).

[17] R. Hinze. Generics for the masses. *J. Funct. Program.*, 16(4-5):451–483, July 2006.

[18] R. Hinze and A. Löh. Generic programming in 3d. *Sci. Comput. Program.*, 74(8):590–628, June 2009.

[19] 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.

[20] R. Lämmel and S. P. Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, TLDI '03, pages 26–37, 2003.

[21] R. Lämmel and S. P. Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *Proceedings of the ninth ACM SIGPLAN international conference on Functional programming*, ICFP '04, pages 244–255, 2004.

[22] S. Peyton Jones, A. Reid, T. Hoare, S. Marlow, and F. Henderson. A semantics for imprecise exceptions. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 25–36, 1999.

[23] T. Sheard and S. P. Jones. Template meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, Haskell '02, pages 1–16, 2002.

A. Proofs

The following are proofs for Section 4. All proofs use the implementation of contracts given in Figure 3.

A.1 General properties of contracts

Lemma A.1. Let `c` be any contract. If `c v = Just v'` implies that `v' ⊑ v`, then `assert c v ⊑ v`.

Proof. Case analysis:

`c v = ⊥`: `assert c v = ⊥ ⊑ v`.
`c v = Just v'`: `assert c v = v' ⊑ v`.
`c v = Nothing`: `assert c v = error "..." = ⊥ ⊑ v`.

□

Lemma A.2. Let `c` be any contract. If `c v = Just v'`, then `v' ⊑ v`.

Proof. Induction on the contract `c`, using its definition.

`⊥ v = Just v'`: Impossible.
`true v = Just v'`: Then `v' = v ⊑ v`.
`false v = Just v'`: Impossible.
`prop p v = Just v'`: Then `v' = v ⊑ v`.
`(c1 & c2) v = Just v'`:
Then `c1 v = Just v''` and `c2 v'' = Just v'`. With ind. hyp. follows `v' ⊑ v'' ⊑ v`.
`(c1 |> c2) v = Just v'`:
Then either `c1 v = Just v'` or `c1 v = Nothing` and `c2 v = Just v'`. With ind. hyp. follows `v' ⊑ v` in either case.
`(c1 >-> c2) v = Just v'`:
If `v = ⊥`, then `v' = ⊥` and thus `v' ⊑ v`.
Otherwise `v' = assert c2 . v . assert c1`. Let `v''` be any possible argument value of the function `v'`. If the first assertion `assert c1 v'' ≠ Just ...` or the second assertion `assert c2 (v (assert c1 v'')) ≠ Just ...`, then `v' v'' = ⊥`. Otherwise by induction hypothesis and Lemma A.1 we have `assert c1 v'' ⊑ v''` and likewise also `assert c2 (v (assert c1 v'')) ⊑ v (assert c1 v'')`. With continuity of all functions follows `v (assert c1 v'') ⊑ v v''`. Alltogether `assert c2 (v (assert c1 v'')) ⊑ v v''`.
Hence in all cases `v' v'' ⊑ v v''` and thus `v' ⊑ v`.
`(pCon c1 ... cn) v = Just v'` where `pCon` is the contract combinator for data constructor `Con`.
From the definition follows that `v = Con v1 ... vn` and `v' = Con (assert c1 v1) ... (assert cn vn)`. By ind. hyp. and Lemma A.1 we get that `assert c1 v1 ⊑ v1`, ..., `assert cn vn ⊑ vn`. So `v' ⊑ v`.

□

Corollary A.3 (Contracts are partial identities)**.**
`assert c v ⊑ v`.

Proof. From Lemma A.2 and Lemma A.1. □

Corollary A.4. `assert c` is strict for any contract `c`.

Proof. From Corollary A.3 follows that `assert c ⊥ ⊑ ⊥`. Hence `assert c ⊥ = ⊥`. □

Lemma A.5 (Asserting &)**.**

```
assert (c1 & c2) = assert c2 . assert c1
```

Proof.

```
  assert (c1 & c2)
=\x -> case c1 x >>= c2 of
          Just z -> z
          Nothing -> error "..."
=\x -> case (case c1 x of
               Just y -> c2 y
               Nothing -> Nothing) of
          Just z -> z
          Nothing -> error "..."
=\x -> case c1 x of
          Just y -> case c2 y of
                      Just z -> z
                      Nothing -> error "..."
          Nothing -> error "..."
=\x -> case c1 x of
          Just y -> assert c2 y
          Nothing -> error "..."
=(assert c is strict, Corollary A.4)
  \x -> case c1 x of
          Just y -> assert c2 y
          Nothing -> assert c2 (error "...")
=(assert c is strict, Corollary A.4)
  \x -> assert c2 (case c1 x of
                     Just y -> y
                     Nothing -> error "...")
=\x -> assert c2 (assert c1 x)
=assert c2 . assert c1
```

□

Claim A.6 (Idempotence of contracts)**.**

```
assert c . assert c = assert c
```

Proof.

```
  assert c . assert c
=(Lemma A.5)
  assert (c & c)
=(claimed idempotence of conjunction)
  assert c
```

□

A.2 Properties of predicate contracts

Lemma A.7 (Disjunction)**.**

```
prop p1 |> prop p2 = prop (\x -> p1 x || p2)
```

Proof.

```
  prop p1 |> prop p2
=\x -> case prop p1 x of
          Just y -> Just y
          Nothing -> prop p2 x
=\x -> case (if p1 x then Just x else Nothing) of
          Just y -> Just y
          Nothing -> prop p2 x
=\x -> if p1 x then Just x else prop p2 x
=\x -> if p1 x then Just x else
          (if p2 x then Just x else Nothing)
=\x -> if p1 x || p2 x then Just x else Nothing
=prop (\x -> p1 x || p2)
```

□

Lemma A.8 (Conjunction)**.**

```
prop p1 & prop p2 = prop (\x -> p1 x && p2 x)
```

Proof.

```
  prop p1 & prop p2
=\x -> prop p1 x >>= prop p2
=\x -> case prop p1 x of
          Just y -> prop p2 y
          Nothing -> Nothing
=\x -> case (if p1 x then Just x else Nothing) of
          Just y -> prop p2 y
          Nothing -> Nothing
=\x -> if p1 x then prop p2 x else Nothing
=\x -> if p1 x
          then (if p2 x then Just x else Nothing)
          else Nothing
=\x -> if p1 x && p2 x then Just x else Nothing
=prop (\x -> p1 x && p2 x)
```

□

A.3 Properties of the conjunction contract combinator `&`

Lemma A.9 (Associativity of `&`)**.**

```
c1 & (c2 & c3) = (c1 & c2) & c3
```

Proof.

```
  c1 & (c2 & c3)
=\x -> c1 x >>= (c2 & c3)
=\x -> c1 x >>= (\y -> c2 y >>= c3)
=(associativity of >>= for the Maybe monad)
  \x -> (c1 x >>= c2) >>= c3
=\x -> (\y -> c1 y >>= c2) x >>= c3
=\x -> (c1 & c2) >>= c3
=(c1 & c2) & c3
```

□

Lemma A.10 (Left neutral element of `&`)**.**

```
true & c = c
```

Proof.

```
  true & c
=\x -> true x >>= c
=\x -> Just x >> c
=(Just = return is left identity for Maybe monad)
=\x -> c x
=c
```

□

Lemma A.11 (Right neutral element of `&`)**.**

```
c & true = c
```

Proof.

```
  c & true
=\x -> c x >>= true
=\x -> c x >> Just
=(Just = return is right identity for Maybe monad)
=\x -> c x
=c
```

□

A.4 Properties of the disjunction contract combinator `|>`

Lemma A.12 (Associativity of `|>`)**.**

```
c1 |> (c2 |> c3) = (c1 |> c2) |> c3
```

Proof.

```
  c1 |> (c2 |> c3)
=\x -> c1 x `mplus` (\y -> c2 y `mplus` c3 y) x
=\x -> c1 x `mplus` (c2 x `mplus` c3 x)
=(associativity of mplus for Maybe)
  \x -> (c1 x `mplus` c2 x) `mplus` c3 x
=\x -> (\y -> c1 y `mplus` c2 y) x `mplus` c3 x
=(c1 |> c2) |> c3
```

□

Lemma A.13 (Left neutral element of `|>`)**.**

```
false |> c = c
```

Proof.

```
  false |> c
=\x -> const Nothing x `mplus` c x
=\x -> Nothing `mplus` c x
=(Nothing = mzero is left identity for Maybe)
  \x -> c x
=c
```

□

Lemma A.14 (Right neutral element of `|>`)**.**

```
c |> false = c
```

Proof.

```
  c |> false
=\x -> c x `mplus` const Nothing x
=\x -> c x `mplus` Nothing
=(Nothing = mzero is right identity for Maybe)
=\x -> c x
=c
```

□

Lemma A.15 (Idempotence of `|>`)**.**

$$\texttt{c |> c} = \texttt{c}$$

Proof.

```
 c |> c
=\x -> c x `mplus` c x
=\x -> case c x of {Just y -> Just y; Nothing -> c x}
=\x -> c x
=c
```

□

A.5 Guarded commutativity and absorption

Lemma A.16 (Guarded commutativity of disjunction)**.**

$$\texttt{c}_1\ \texttt{|>}\ (\texttt{c}_2\ \texttt{|>}\ \texttt{c}_1) = \texttt{c}_1\ \texttt{|>}\ \texttt{c}_2$$

Proof.

```
  c1 |> (c2 |> c1)
=\x -> case c1 x of
         Just y -> Just y
         Nothing -> case c2 x of
                      Just y -> Just y
                      Nothing -> c1 x
=(case distinction c1 x = ⊥, Just y, Nothing)
 \x -> case c1 x of
         Just y -> Just y
         Nothing -> case c2 x of
                      Just y -> Just y
                      Nothing -> Nothing
=\x -> case c1 x of
         Just y -> Just y
         Nothing -> c2 x
=c1 |> c2
```

□

Lemma A.17 (Absorption 1)**.**

$$\texttt{c}_1\ \texttt{|>}\ (\texttt{c}_1\ \texttt{\&}\ \texttt{c}_2) = \texttt{c}_1$$

Proof.

```
  c1 |> (c1 & c2)
=\x -> case c1 x of
         Just y -> Just y
         Nothing -> case c1 x of
                      Just z -> c2 z
                      Nothing -> Nothing
=(case distinction c1 x = ⊥, Just y, Nothing)
 \x -> case c1 x of
         Just y -> Just y
         Nothing -> Nothing
=c1
```

□

A.6 Corollaries

Corollary A.18.

$$\texttt{false \& c} = \texttt{false}$$

Proof.

```
  false & c
=(Lemma A.13, left neutral element of |>)
  false |> (false & c)
=(Lemma A.17, absorption 1)
  false
```

□

Corollary A.19.

$$\texttt{true |> c} = \texttt{true}$$

Proof.

```
  true |> c
=(Lemma A.10, left neutral element of &)
  true |> (true & c)
=(Lemma A.17, absorption 1)
  true
```

□

A.7 Properties of the function contract combinator `>->`

Lemma A.20 (Relation with `true`)**.**

$$\texttt{true >-> true} = \texttt{true}$$

Proof.

```
 true >-> true
=\f -> Just (f `seq` (assert true . f . assert true))
=\f -> Just (f `seq` (id . f . id))
=\f -> Just f
=true
```

□

Lemma A.21 (`false` postcondition)**.**

$$\texttt{c}_1\ \texttt{>->}\ \texttt{false} = \texttt{c}_2\ \texttt{>->}\ \texttt{false}$$

Proof.

```
  c1 >-> false
=\f -> Just (f `seq` (assert false . f . assert c1))
=\f ->
    Just (f `seq` (const (error "...") . f . assert c1))
=\f -> Just (f `seq` const (error "..."))
=\f ->
    Just (f `seq` (const (error "...") . f . assert c2))
=\f -> Just (f `seq` (assert false . f . assert c2))
=c2 >-> false
```

□

Lemma A.22 (Distribution with `&`)**.**

$$(c_1 \texttt{ >-> } c_2) \texttt{ \& } (c_3 \texttt{ >-> } c_4) = (c_3 \texttt{ \& } c_1) \texttt{ >-> } (c_2 \texttt{ \& } c_4)$$

Proof.

```
  (c1 >-> c2) & (c3 >-> c4)
=\f -> (c1 >-> c2) f >>= (c3 >-> c4)
=\f -> Just (f `seq` (assert c2 . f . assert c1))
        >>= (c3 >-> c4)
=\f -> (c3 >-> c4) (f `seq` (assert c2 . f . assert c1))
=\f -> Just (f `seq` (assert c4 . assert c2 . f .
                      assert c1 . assert c3))
=(Lemma A.5)
  \f -> Just (f `seq` (assert (c2 & c4) . f .
                      assert (c3 & c1)))
=(c3 & c1) >-> (c2 & c4)
```

□

Lemma A.23 (Non-distribution with `|>`)**.**

$$(c_1 \texttt{ >-> } c_2) \texttt{ |> } (c_3 \texttt{ >-> } c_4) = c_1 \texttt{ >-> } c_2$$

Proof.

```
  (c1 >-> c2) |> (c3 >-> c4)
=\f -> (c1 >-> c2) f `mplus` (c3 >-> c4) f
=\f -> Just (f `seq` (assert c2 . f . assert c1))
        `mplus` (c3 >-> c4) f
=(MonadPlus law for Just = return)
  \f -> Just (f `seq` (assert c2 . f . assert c1))
=c1 >-> c2
```

□

B. Properties of the contract combinator `io`

The following are proofs for Section 7.2.

Lemma B.1.

$$\texttt{io } c_1 \texttt{ \& io } c_2 = \texttt{io } (c_1 \texttt{ \& } c_2)$$

Proof.

```
  io c1 & io c2
=\x -> io c1 x >>= io c2
=\x -> Just (x >>= return . assert c1) >>=
            \y -> Just (y >>= return . assert c2)
=\x -> Just ((x >>= return . assert c1) >>=
             return . assert c2)
=\x -> Just (x >>= (return . assert c1 >>=
                    return . assert c2))
=\x -> Just (x >>= (return . assert c2 . assert c1))
=\x -> Just (x >>= (return . assert (c1 & c2))
=io (c1 & c2)
```

□

Lemma B.2.

$$\texttt{io } c_1 \texttt{ |> io } c_2 = \texttt{io } c_1$$

Proof.

```
  io c1 |> io c2
=\x -> io c1 x `mplus` io c2 x
=\x -> Just (x >>= return . assert c1)
        `mplus` io c2 x
=\x -> Just (x >>= return . assert c1)
=io c1
```

□

Lemma B.3.

$$\texttt{io true} = \texttt{true}$$

Proof.

```
  io true
=\x -> Just (x >>= return . assert true)
=\x -> Just (x >>= return . id)
=\x -> Just (x >>= return)
=\x -> Just io
=true
```

□

C. Two definitions of the list contract combinator

The two list contract combinators are not equal, but we can prove that they are equal in all contract contexts surrounded by `assert`. Basically the results $\bot$ and `Just` $\bot$ of a contract cannot be distinguished. We denote this equivalence by $\equiv$.

The next lemma is not only of interest for the subsequent lemma. However, the two expressions will give different fault explanations if a contract is violated.

Lemma C.1.

$$\texttt{assert (list c) } V = \texttt{map (assert c) } V$$

Proof. Structural induction on the value V.

$V = \bot$:

```
  assert (list c) ⊥
=assert (pNil |> pCons c (list c)) ⊥
=⊥
=map (assert c) ⊥
```

$V =$ `[]`:

```
  assert (list c) []
=assert (pNil |> pCons c (list c)) []
=[]
=map (assert c) []
```

$V = W{:}WS$:

```
  assert (list c) (W:WS)
=assert (pNil |> pCons c (list c)) (W:WS)
=assert c W : assert (list c) WS
=(induction hypothesis)
  assert c W : map (assert c) WS
=map (assert c) (W:WS)
```

□

Lemma C.2.

```
list c V ≡ list' c V
```

Proof. Structural induction on the value V.

$V = \bot$:

```
  list c ⊥
=(pNil |> pCons c (list c)) ⊥
=⊥
≡Just ⊥
=list' c ⊥
```

$V =$ `[]`:

```
  list c []
=(pNil |> pCons c (list c)) []
=Just []
=Just (map (assert c) [])
=list' c []
```

$V = W{:}WS$:

```
  list c (W:WS)
=(pNil |> pCons c (list c)) (W:WS)
=Just (assert c W : assert (list c) WS)
=(Lemma C.1)
  Just (assert c W : map (assert c) WS)
=Just (map (assert c) (W:WS))
=list' c (W:WS)
```

□

D. Some Properties that do *not* hold

Contracts do not form a distributive lattice with `&` and `|>`. So a number or properties that might be expected do not actually hold. Comparing with the properties of the lazy Booleans is helpful in identifying which properties hold and which do not.

Lemma D.1 (Commutativity)**.**
`&` is not commutative.

Proof. For example

```
⊥ & false = \x -> ⊥
false & ⊥ = \x -> Nothing
```

So

```
(⊥ & false) |> true = \x -> ⊥
(false & ⊥) |> true = \x -> Just x
```

Hence

```
assert ((⊥ & false) |> true) e = ⊥
assert (false & ⊥) |> true) e = e
```

□

Similarly `&&` is not commutative:

```
⊥ && False = ⊥ ≠ False = False && ⊥
```

Lemma D.2 (Commutativity)**.**
`|>` is not commutative.

Proof. For example

```
⊥ |> true = \x -> ⊥
true |> ⊥ = \x -> Just x
```

Hence

```
assert (⊥ |> true) e = ⊥
assert (true |> ⊥) e = e
```

□

Similarly `||` is not commutative.

```
⊥ || True = ⊥ ≠ True = True || ⊥
```

Lemma D.3 (Distributivity 1)**.**

```
(c1 & c2) |> c3 ≠ (c1 |> c3) & (c2 |> c3)
```

Proof.

```
(false |> true) & (⊥ |> true) = \x -> ⊥
          (false & ⊥) |> true = \x -> Just x
```

□

Note that also

```
(b1 || b2) && b3 ≠ (b1 && b3) || (b2 && b3)
```

because

```
  (False || True) && (⊥ || True)
=⊥
⊏True
=(False && ⊥) || True
```

Lemma D.4 (Distributivity 2)**.**

```
(c1 & c3) |> (c2 & c3) ≠ (c1 |> c2) & c3)
```

Proof.

```
(true & false) |> (⊥ & false) = \x -> ⊥
          (true |> ⊥) & false = \x -> Nothing
```

□

Note that also

$$\texttt{(b}_1\texttt{ \&\& b}_3\texttt{) || (b}_2\texttt{ \&\& b}_3\texttt{)} \neq \texttt{(b}_1\texttt{ || b}_2\texttt{) \&\& b}_3$$

because

$$\begin{aligned}
&\texttt{(True \&\& False) || (}\bot\texttt{ \&\& False)}\\
=&\bot\\
\sqsubset&\texttt{False}\\
=&\texttt{(True || }\bot\texttt{) \&\& False}
\end{aligned}$$

Lemma D.5.

$$\texttt{false >-> false} \neq \texttt{false}$$

Proof.

$$\begin{aligned}
&\texttt{(false |> true) f} = \texttt{f}\\
&\texttt{((false >-> false) |> true) f} = \texttt{const (error "...")}
\end{aligned}$$

□