# A Semantics for Lazy Assertions

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

## Abstract

Lazy functional programming languages need lazy assertions to ensure that assertions preserve the meaning of programs. Examples in this paper demonstrate that previously proposed lazy assertions nonetheless break basic semantic equivalences, because they include a non-deterministic disjunction combinator. The objective of this paper is to determine "correct" definitions for lazy assertions. The starting point is our formalisation of basic properties such as laziness, taking them as axioms of our design space. We develop the first denotational semantics for lazy assertions; assertions denote subdomains. We define a weak disjunction combinator and together with a conjunction combinator assertions form a bounded distributive lattice. From the established laws we derive an efficient prototype implementation of lazy assertions for Haskell as a library.

***Categories and Subject Descriptors*** D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; F.3.2 [*Semantics of Programming Languages*]: Denotational semantics

***General Terms*** Languages, Reliability, Theory

## 1. Introduction

Assertions have been used by programmers for a long time to express within their programming language properties of parts of their programs [20]. Assertions document expected properties. They are usually checked at runtime; if a fault occurs, the assertion raises an exception and thus helps locating the cause of the fault [21].

Assertions look like the perfect match for lazy functional programming languages: Because assertions are side-effect free functions, inserting or removing an assertion should not change the meaning of a program, except when the violation of an assertion is reported by an exception. Let us consider a simple example of an assertion in Haskell:

```
assert nats [4,2]
```

Here `nats` shall be an assertion expressing that a given list is a list of natural numbers. The function `assert` applies the assertion[1] to the list `[4,2]`. The expression evaluates to `[4,2]`. In contrast, the expression

```
assert nats [4,-2]
```

raises an exception, because the assertion is violated. The function `assert nats` is a partial identity on integer lists.

A lazy functional language allows the definition of infinite data structures. We should be allowed to insert our assertion directly into the recursive definition of the infinite list of Fibonacci numbers:

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

Here `zipWith` combines the Fibonacci list and the Fibonacci list without its first element via element-wise addition.

For this definition to work, just as it does without the application of `assert nats`, the assertion must be lazy. An assertion is lazy, if the argument of assertion application is evaluated always only as far as it is demanded by the context of the assertion application. In denotational terms this means that the assertion has to accept not just total lists of natural numbers, but also any partial approximation of such a list:

```
assert nats (0:1:⊥)       ⇝  0:1:⊥
assert nats (0:1:1:⊥)     ⇝  0:1:1:⊥
assert nats (0:1:1:2:⊥)   ⇝  0:1:1:2:⊥
```

In general, any approximation of an acceptable value has to be accepted by an assertion to ensure that the programmer can insert assertions anywhere without changing the meaning of the program. Lazy languages need lazy assertions.

Several lazy assertion systems have been proposed [5–7], but they turn out to be unacceptable, because they break basic semantic equivalences that are used by any optimising compiler. Consider an assertion `equal` for tuples of Booleans which asserts that the two values of the tuple are equal. This assertion can easily be defined in all previously proposed systems. As expected, the program snippet

```
let x = assert equal (True,False)
in (fst x, snd x)
```

raises an exception when evaluated. However, let us apply the basic program transformation of let-inlining and we get

```
(fst (assert equal (True,False)),
 snd (assert equal (True,False)))
```

Evaluating this snippet yields `(True,False)`. No assertion exception is raised! The simple reason is that each of the two assertions only sees a partial value and both `(True,⊥)` and `(⊥,False)` have to be accepted by a lazy assertion as they are approximations of the

[1] We follow common practise of separating the assertion and the assertion application function `assert`. In practise `assert` also takes additional arguments to name the error location in case the assertion is violated. Some assertion systems [9] fuse `assert` with the assertion though refunctionalisation, which can improve performance. However, considering assertions as data that is interpreted by `assert` makes the implementation design easier, because definitions of interpreters are generally easier to understand than higher-order functions.



*PEPM'11,* January 24–25, 2011, Austin, Texas, USA.

evidently acceptable values `(True,True)` and `(False,False)` respectively.

The nub of the problem is that `assert equal` is not a function, it is non-deterministic. The value of `assert equal (True,False)` is either `(True, error "assertion violated")` or `(error "assertion violated", False)`, depending on which part of the tuple is demanded first. Previously proposed assertion systems use non-pure extensions such as `unsafePerformIO` to step outside the pure functional language to enable the implementation of `assert equal`. Allowing such "non-deterministic functions" whose results depend on the context requires a complex, non-standard language semantics. As we have just seen, it also breaks standard equivalences.

We do not want to repeat the experience of first implementing a new assertion system and later finding that it does not meet basic semantic requirements. Instead we start with defining a simple denotational semantics, ensuring that lazy assertions will have the expected semantics. We formalise requirements such as laziness and take them as axioms that determine our design space. From the axioms we derive many further properties of lazy assertions. In particular we establish that lazy assertions with a conjunctive and a disjunctive combinator form a bounded distributive lattice. We see that disjunction has to be weaker than we might expect and that assertions do not allow for a general negation operator. Only after establishing these general properties we finally consider assertions for specific type structures, mainly algebraic data types. We use the previously proved properties to derive an efficient implementation in Haskell.

The example of Boolean tuples demonstrated that proper lazy assertions by their very nature cannot express properties that relate different parts of a data structure. Hence we start in Section 2 with a number of examples showing that despite this limitation many useful applications of the lazy assertions, which we derive in this paper, exist. Then Section 3 sets out our denotational semantics and motivates and formalises axioms. Subsequently Section 4 derives central properties of lazy assertions and develops a less intuitive, but smaller set of axioms for future use. Section 5 is the core of the paper, defining general assertion combinators and proving their properties. In Section 6 we then instantiate the general semantics for specific data types, prove type-specific laws and derive the implementation. Section 7 discusses related work and Section 8 concludes.

## 2. Examples for Assertions

Which kind of properties can we express with lazy assertions? For primitive built-in types such as `Integer` and `Char` we have a combinator that allows any Boolean predicate to check the desired property:

```
pred :: Prim t => (t -> Bool) -> Assert t
```

For algebraic data types lazy assertions are mainly patterns. For the list data type with its two constructors

```
(:) :: t -> [t] -> [t]
[]  :: [t]
```

we have two corresponding assertion combinators

```
pCons :: Assert t -> Assert [t] -> Assert [t]
pNil  :: Assert [t]
```

Furthermore, the assertion combinator

```
(<|>) :: Assert t -> Assert t -> Assert t
```

combines two assertions disjunctively, a value may fulfil the first or the second assertion. With these combinators we can define the assertion of the introduction:

```
nats :: Assert [Integer]
nats = pNil <|> pCons (pred (>=0)) nats
```

So a list of natural numbers is either an empty list or a non-empty list with a natural number as first element and a list of natural numbers as the tail. The definition reads similar to an algebraic data type definition in Haskell.

Many language features are useful for better defining assertions. For example, we may want to separate the reusable idea of asserting a property for all elements of a list:

```
nats :: Assert [Integer]
nats = pList (pred (>=0))

pList :: Assert t -> Assert [t]
pList a = pNil <|> pCons a (pList a)
```

We also have a lazy assertion combinator

```
pAny :: Assert t
```

that accepts any value of any type `a`.

We can easily define an assertion that accepts only infinite lists:

```
infinite :: Assert [t]
infinite = pCons pAny infinite
```

The assertion accepts also all partial approximations, such as $7:\bot$, but not any list ending with the constructor `[]`.

We can also define assertion combinators with parameters that are not assertions. For example, we can define assertions that only accept lists of a given minimal length:

```
lengthAtLeast :: Int -> Assert [t]
lengthAtLeast 0     = pAny
lengthAtLeast (n+1) = pCons pAny (lengthAtLeast n)
```

If the given length is 0, any list is accepted. If it is greater, then the list must be non-empty and the tail must be of a length 1 shorter. This assertion might be used in a definition such as

```
initAv :: [Int] -> Int
initAv = assert (lengthAtLeast 5 |-> pAny) initAv'

initAv' xs = sum (take 5 xs) 'div' 5
```

Here

```
(|->) :: Assert s -> Assert t -> Assert (s->t)
```

wraps assertions for a pre- and a post-condition around a function. The assertion `lengthAtLeast` guarantees that `initAv` always determines the average of the first 5 list elements; it raises an exception if the list is too short.

The assertion `lengthAtLeast` is strict in its numeric argument. Using it with a constant argument is safe, but otherwise it could turn a terminating into a non-terminating program.

Let us consider a bigger example. Assume we have a program manipulating propositional formulae with functions computing various normal forms. The data type for representing a propositional formula is

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

Now we want to assert that some function produces a conjunctive normal form. The assertion `conjNF` defined below accepts the set of conjunctive normal forms:

```
conjNF, disj, lit, atom :: Assert Form

conjNF = pAnd conjNF conjNF <|> disj
disj = pOr disj disj <|> lit
```

```
lit = pNot atom <|> atom
atom = pAtom pAny
```

Again these definitions look similar to several algebraic data type definitions or a context-free grammar.

Another function may transform a formula such that all binary operations are left-associated. The assertion `left` can check for this property:

```
left, noImp, noAnd, noOr, nonBind :: Assert Form

left = noImp <|> noAnd <|> noOr
noImp = pAnd left noAnd <|> pOr left noOr <|> nonBin
noAnd = pImp left noImp <|> pOr left noOr <|> nonBin
noOr = pImp left noImp <|> pAnd left noAnd <|> nonBin
nonBin = pNot left <|> pAtom pAny
```

These definitions contain repeated code. Easier to read is an equivalent definition that uses the combinator `<\>` that subtracts a given constructor from the set of accepted values, that is, any term with the given constructor at the top is not accepted.

```
left = pImp left (left <\> cImp) <|>
       pAnd left (left <\> cAnd) <|>
       pOr left (left <\> cOr) <|>
       pNot left <|> pAtom pAny
```

If we want to ensure that a formula is both in conjunctive normal form and all binary operators left-associated, we just use the conjunction combinator `<&>`:

```
leftConjNF :: Assert Form
leftConjNF = conjNF <&> left
```

## 3. Semantics and Axioms

Every assertion asserts a property for values of a given domain, such as the values of type `Form`. For our denotational semantics of lazy assertions we make the standard assumption that a domain $D$ is a directed complete partial order with least element $\bot$ [1] . So every directed subset has a least upper bound. For simplicity we also assume that the exception raised by assertion violation is represented by the least element $\bot$ of the domain; it is not distinct.

What exactly is an assertion? An assertion $a$ divides the domain $D$ into a set of accepted values $[\![a]\!]$ and the remaining non-accepted values. The function `assert` turns any assertion $a$ into a function that maps from the domain into the domain. From now on we write $\langle a\rangle : D \to D$ for the semantics of the expression `assert` $a$. From $\langle a\rangle$ we obtain the acceptance set $[\![a]\!]$:

DEFINITION 1 (Acceptance set of an assertion).
*Let $a$ be an assertion. Then*

$$[\![a]\!] := \{v \in D \mid \langle a\rangle\, v = v\}$$

*is the* acceptance set of $a$.

Our axioms for lazy assertions are the following:

DEFINITION 2 (Axioms for lazy assertions).
*$a$ is a* lazy assertion*, if it meets the following axioms:*

1. *$\langle a\rangle : D \to D$ is a continuous function.*
2. *$a$ is* trustworthy*, that is, $\langle a\rangle\, v \in [\![a]\!]$ for any value $v$.*
3. *$\langle a\rangle$ is a* partial identity*, that is, $\langle a\rangle\, v \sqsubseteq v$ for any value $v$.*
4. *$[\![a]\!]$ is a lower set.*

A lower set is a set that contains all approximations of its elements, that is, $S$ is lower if and only if $v \in S$ and $v' \sqsubseteq v$ imply $v' \in S$.

We require the four axioms for the following reasons:

1. We want to ensure that the addition of assertions does not change the semantics of the lazy language in any way. Hence we require $\langle a\rangle$ to be a function. To be sure and for ease of implementation we also aim to define lazy assertions as a library within the lazy language itself. Consequently $\langle a\rangle$ has to be a continuous function.
2. A purpose of an assertion is to protect the context of the assertion-wrapped expression, and thus the remainder of the programme, from ever seeing a value that is unacceptable for the assertion. Trustworthiness captures this property.
3. To preserve the semantics of a program when introducing assertions, an assertion has to be the identity on all values of the acceptance set. Even for values outside the acceptance set an assertion should not return an arbitrary value, but some approximation of the original value.
4. We already argued in the introduction, that any approximation $v'$ of a value $v$ that is evidently acceptable must be accepted as well. When the context of the assertion application forces evaluation only up to $v'$, the demanded parts of $v'$ have to be returned. Raising an exception would be wrong, because $v'$ may later be evaluated further to $v$. When only $v'$ is demanded the assertion may not evaluate further to check whether $v'$ is actually $v$, because that would be eager evaluation and could cause non-termination.

Any acceptance set of a lazy assertion is a domain. Trivially, any subset $A$ of a domain $D$ is a partial order and the least upper bound of any directed subset in $D$ is also a least upper bound in $A$. However, we have to prove that these least upper bounds are in $A$. We also need $\bot \in A$.

THEOREM 1 (Acceptance sets are domains).
*If $a$ is a lazy assertion, then $[\![a]\!]$ is a domain.*

**Proof:** Let $X \subseteq [\![a]\!]$ be directed. With the definition of $[\![a]\!]$ follows $\langle a\rangle\, X = X$; hence $\bigsqcup(\langle a\rangle\, X) = \bigsqcup X$. Because $\langle a\rangle$ is continuous, $\langle a\rangle\,(\bigsqcup X) = \bigsqcup(\langle a\rangle\, X)$. Both together give $\langle a\rangle\,(\bigsqcup X) = \bigsqcup X$. Hence $\bigsqcup X \in [\![a]\!]$.

There is at least one element to which all assertion arguments are mapped. Partial identity tells us that $\bot$ must be mapped to $\bot$ (in that sense our lazy assertions are strict), and hence any acceptance set must contain $\bot$. □

So assertions really describe subtypes; their acceptance sets are subdomains.

Trustworthiness is clearly desirable, but it is just a different way of stating that the assertion function should be idempotent:

LEMMA 2 (Trustworthiness and idempotency).
*$\forall v \in D.\ \langle a\rangle\, v \in [\![a]\!]$ if and only if $\forall v \in D.\ \langle a\rangle\,(\langle a\rangle\, v) = \langle a\rangle\, v$.*

**Proof:** Because $[\![a]\!] = \{w \mid \langle a\rangle\, w = w\}$. □

Findler and Blume [9] previously proposed projections as a semantics for contracts.

DEFINITION 3. *A function $p : D \to D$ on a domain $D$ is a* projection *if*

- *it is continuous,*
- *it is idempotent, and*
- *it is a partial identity.*

LEMMA 3 (The image of a projection is an acceptance set).
*If $p$ is a projection, then $\{p\, v \mid v \in D\} = \{v \in D \mid p\, v = v\}$.*

**Proof:** Let $v' \in \{p\, v \mid v \in D\}$. Hence exists $\hat{v} \in D$ with $v' = p\,\hat{v}$. Idempotency gives $p\,(p\,\hat{v}) = p\,\hat{v}$. Hence $v' = p\,\hat{v} \in \{v \in D \mid p\, v = v\}$.

Let $v' \in \{v \in D \mid p\, v = v\}$. So $p\, v' = v'$. Hence $v' \in \{p\, v \mid v \in D\}$. □

So if $\langle a \rangle$ is a projection, then its image is $[\![a]\!]$. Together with Lemma 2 we conclude:

THEOREM 4 (Assertions are lower projections).
*$a$ is a lazy assertion if and only if $\langle a \rangle$ is a projection and its image is a lower set.*

Our axioms do not explicitly demand that assertion application raises an exception for unacceptable values. However, it is a consequence of the axioms that the result is a partial value and thus "contains" $\bot$, which represents our exception.

LEMMA 5 (Assertions are prompt). *$\langle a \rangle\, v \sqsubset v$, for all $v \notin [\![a]\!]$.*

**Proof:** Let $v \notin [\![a]\!]$. Because $\langle a \rangle$ is a partial identity, $\langle a \rangle\, v \sqsubseteq v$. Because $v \notin [\![a]\!]$, but according to trustworthiness $\langle a \rangle\, v \in [\![a]\!]$, we get $\langle a \rangle\, v \sqsubset v$. □

## 4. Revised Axioms

We already have two characterisations of lazy assertions: Through the well-motivated axioms of Definition 2 and as lower projections (Theorem 4). Now we aim to determine an alternative set of axioms which require us to check fewer properties and that will make it easier to find concrete assertions.

We cannot simply drop any of our axioms. None is implied by the others, for each axiom we can easily construct some pseudo-assertion that does not meet this axiom but all others. Instead we note that we defined the acceptance set $[\![a]\!]$ of an assertion through assertion application $\langle a \rangle$. We consider doing the opposite: Given a suitable acceptance set $[\![a]\!]$, its assertion application $\langle a \rangle$ is uniquely determined.

First we define the lower set for a given value $v$ and a set $A \subseteq D$ with respect to a given value:

$$\downarrow\{v\} := \{v' \mid v' \sqsubseteq v\}$$
$$A_v := \downarrow\{v\} \cap A$$

With these we define assertion application:

THEOREM 6. *Let $a$ be a lazy assertion. For any value $v \in D$*

$$\langle a \rangle\, v \in [\![a]\!]_v \quad \textit{and} \quad \langle a \rangle\, v = \bigsqcup [\![a]\!]_v$$

**Proof:** Partial identity yields $\langle a \rangle\, v \sqsubseteq v$ and trustworthiness $\langle a \rangle\, v \in [\![a]\!]$. Together we have $\langle a \rangle\, v \in \downarrow\{v\} \cap [\![a]\!] = [\![a]\!]_v$.

Let $w \in [\![a]\!]_v$. So $w \sqsubseteq v$ and with monotonicity follows $\langle a \rangle\, w \sqsubseteq \langle a \rangle\, v$. Also $w \in [\![a]\!]$ and thus $\langle a \rangle\, w = w$. Together we get $w \sqsubseteq \langle a \rangle\, v$ and hence $\langle a \rangle\, v$ is an upper bound for $[\![a]\!]_v$. Because $\langle a \rangle\, v \in [\![a]\!]_v$, it is the least upper bound of $[\![a]\!]_v$. □

We now have a complete definition for assertion application. Our axioms did not leave us any freedom of choice. But we do not have yet a complete definition of assertions, because there is an infinite number of sets we might choose as acceptance set, and for each of these we need to check our axioms.

So let us examine acceptance sets more closely.

COROLLARY 7. *$[\![a]\!]_v$ is an ideal for any value $v$.*

**Proof:** As the intersection of two lower sets, $[\![a]\!]_v$ is a lower set. Because $[\![a]\!]_v$ contains an upper bound for all elements according to Theorem 6, it is directed. □

So $[\![a]\!]_v$ has to be an ideal, but $[\![a]\!]$ does *not* have to be an ideal! For an example of an acceptance set that is not an ideal, consider an assertion that accepts all values of a domain. It trivially meets all our axioms. Domains are generally not directed. For instance, the domain of Boolean values contains both `True` and `False` which have no common upper bound.

Do we really need to check all our axioms for any assertion we come up with? Indeed, we only have to ensure that our acceptance set meets certain properties:

DEFINITION 4 (Lazy domain). *A set $A \subseteq D$ is a* lazy domain *if*

- *$A$ is lower,*
- *$A$ contains the least upper bound of any directed subset, and*
- *$A_v$ is directed for all values $v \in D$.*

For any lazy assertion $a$ its acceptance set $[\![a]\!]$ is a lazy domain, but also the reverse is true:

THEOREM 8 (A lazy domain determines an assertion). *Let $A \subseteq D$ be a lazy domain. Let assertion application be defined as*

$$\langle a \rangle\, v := \bigsqcup A_v$$

*Then $a$ is a lazy assertion with acceptance set $[\![a]\!] := A$.*

**Proof:** First we show $A = [\![a]\!]$.

Let $v \in A$. Then $\downarrow\{v\} \cap A = \downarrow\{v\}$. Hence $\langle a \rangle\, v = \bigsqcup A_v = \bigsqcup \downarrow\{v\} = v$. So $v \in [\![a]\!]$.

Let $v \in [\![a]\!]$. So $\langle a \rangle\, v = v$. Thus $\bigsqcup A_v = v$. Because $A_v$ is directed and $A$ contains the least upper bound of any directed subset, $v \in A$.

Next we check the assertion axioms:

1. Because $\langle a \rangle$ is defined through the continuous operations lower set, intersection with a fixed set and least upper bound, it is continuous itself.
2. Trustworthiness: Because $A_v$ is directed and $A$ contains the least upper bound of any directed subset, $\langle a \rangle\, v = \bigsqcup A_v \in A$.
3. Partial identity: Let $v \in D$ be any value. $v$ is an upper bound of $\downarrow\{v\}$. Thus it is an upper bound of $\downarrow\{v\} \cap A$. Hence the least upper bound $\langle a \rangle\, v$ is less or equal the upper bound $v$.

□

So in this section we have shown that our old, intuitive axioms are equivalent to fewer, less intuitive axioms, which however are easier to check. In the future we just make sure that we chose acceptance sets that are lazy domains.

## 5. Assertion Combinators

For any value $v$ we can easily define an assertion called $v$ that accepts $v$ and all its approximations:

$$[\![v]\!] := \{w \mid w \sqsubseteq v\}$$

This acceptance set is a lazy domain and thus gives us a lazy assertion. However, what we really are looking for is a set of combinators for building interesting assertions from very simple ones. These combinators should have a useful algebra.

### 5.1 Minimal and maximal assertions

The canonical partial ordering of assertions is the subset relationship of their acceptance set.

All acceptance sets are domains. Hence the minimal acceptance set contains just the least element $\bot$. The assertion that accepts any value of the domain is trivially the maximal assertion.

$$[\![\texttt{pNone}]\!] = \{\bot\}$$
$$[\![\texttt{pAny}]\!] = D$$

These two acceptance sets are lazy domains. Implementing assertion application for either assertion is straightforward:

$$\langle \texttt{pNone} \rangle\, v = \bigsqcup [\![\texttt{pNone}]\!]_v = \bigsqcup \{\bot\} = \bot$$
$$\langle \texttt{pAny} \rangle\, v = \bigsqcup [\![\texttt{pAny}]\!]_v = \bigsqcup \downarrow\{v\} = v$$

For `pNone` the assertion function constantly returns $\bot$, for `pAny` it is the identity.

### 5.2 Conjunction

The definition for conjunction of two assertions is straightforward:

DEFINITION 5 (Conjunction of assertions).

$$[\![a \texttt{<\&>} b]\!] := [\![a]\!] \cap [\![b]\!]$$

Conjunction is well-defined, because the intersection of two lower sets $[\![a]\!]$ and $[\![b]\!]$ is a lower set, intersection preserves least upper bounds, and the intersection of two directed sets is a directed set, so

$$\begin{aligned}[\![a \texttt{<\&>} b]\!]_v &= {\downarrow}\{v\} \cap ([\![a]\!] \cap [\![b]\!]) \\ &= ({\downarrow}\{v\} \cap [\![a]\!]) \cap ({\downarrow}\{v\} \cap [\![b]\!]) \\ &= [\![a]\!]_v \cap [\![b]\!]_v\end{aligned}$$

is directed.

LEMMA 9. *Conjunction of assertions is commutative and associative and has the assertion `pAny` as neutral element.*

**Proof:** The defining intersection operator is commutative and associative with the full set (domain) as neutral element. □

LEMMA 10. *Conjunction equals two assertions* $\langle a \texttt{<\&>} b\rangle\, v = \langle a\rangle\,(\langle b\rangle\, v)$ *for any value* $v$.

**Proof:**

$$\begin{aligned}&\langle a\rangle\,(\langle b\rangle\, v) \\ =&\text{(definition)} \\ &\langle a\rangle\,(\bigsqcup({\downarrow}\{v\} \cap [\![b]\!])) \\ =&(\langle a\rangle \text{ continuous}) \\ &\bigsqcup\{\langle a\rangle\, w \mid w \in ({\downarrow}\{v\} \cap [\![b]\!])\} \\ =&\text{(definition)} \\ &\bigsqcup\{\bigsqcup({\downarrow}\{w\} \cap [\![a]\!]) \mid w \in ({\downarrow}\{v\} \cap [\![b]\!])\} \\ =&({\downarrow}\{v\} \cap [\![b]\!] \text{ is already lower}) \\ &\bigsqcup({\downarrow}\{v\} \cap [\![b]\!] \cap [\![a]\!]) \\ =&\text{(definition)} \\ &\langle a \texttt{<\&>} b\rangle\, v\end{aligned}$$

□

### 5.3 Disjunction

The cause of the semantic problems of previous lazy assertion contract systems is the presence of standard disjunction in these systems. Disjunction in these systems cannot be given a simple denotational semantics, because it is non-deterministic. Here we see that defining

$$[\![a \vee b]\!] := [\![a]\!] \cup [\![b]\!]$$

does not work, because in general

$$\begin{aligned}[\![a \vee b]\!]_v &= {\downarrow}\{v\} \cap ([\![a]\!] \cup [\![b]\!]) \\ &= ({\downarrow}\{v\} \cap [\![a]\!]) \cup ({\downarrow}\{v\} \cap [\![b]\!])\end{aligned}$$

is not directed, because ideals are not closed under unions, and this union hence generally does not have a least upper bound.

Defining a disjoint disjunction is an option. If $[\![a]\!]$ and $[\![b]\!]$ are disjoint, except for the element $\bot$, then $({\downarrow}\{v\} \cap [\![a]\!]) \cup ({\downarrow}\{v\} \cap [\![b]\!])$ is directed and thus an ideal. For most of our examples in Section 2 disjoint union would be sufficient. The main problem is that there is no easy way to statically (e.g. in the type system) ensure that the acceptance sets of two assertions are disjoint and partially defined operations complicate any algebra.

Instead we take inspiration from ideal completion. The least ideal containing a given set $X$ is given by intersecting all ideals that contain the set $X$: $\bigcap\{Y \mid X \subseteq Y, Y \text{ is ideal}\}$. Similarly we define a new disjunction operator `<|>`:

DEFINITION 6 (Disjunction of assertions).

$$[\![a \texttt{<|>} b]\!] := \bigcap\{Y \mid [\![a]\!] \cup [\![b]\!] \subseteq Y, Y \textit{ lazy domain}\}$$

There always exists at least one $Y$ fulfilling the required properties, namely the domain itself. In the extreme case the disjoint union returns `pAny`.

Our disjunction combinator is well-defined: $[\![a \texttt{<|>} b]\!]$ is a lazy domain, because it is the intersection of lazy domains $Y$. In particular, $[\![a \texttt{<|>} b]\!]_v$ is directed for any $v \in D$ by definition.

Let us consider a few simple examples. Let `fstTrue` and `sndTrue` be assertions on Boolean tuples that determine that the first, respectively the second component of a tuple is `True`. The assertion `fstTrue <|> sndTrue` accepts any tuple! That may be surprising, but if we expected it to reject `(False,False)`, what result should it return for `(False,False)`? Both `(`$\bot$`,False)` and `(False,`$\bot$`)` are suitable candidates in the acceptance set. Neither of them is better than the other, we need a common upper bound and hence `(False,False)` was added to the acceptance set.

Note that `fstTrue <&> sndTrue` is the assertion that requires both components of a tuple to be `True`. Similarly we can define a `fstFalse <&> sndFalse`. Combining these disjunctively, `(fstTrue <&> sndTrue) <|> (fstFalse <&> sndFalse)`, again yields the assertion that accepts *every* tuple. We cannot express that the two components of the tuple should be equal. Relationships between components are not expressible.

LEMMA 11. *Disjunction of assertions is commutative and associative and has the assertion `pNone` as neutral element.*

**Proof:** The union operator in the definition of disjunction is commutative and associative with the set $\{\bot\}$ (included in any acceptance set) as neutral element. □

### 5.4 Conjunction & Disjunction

LEMMA 12 (Absorption laws).

$$\begin{aligned}a \texttt{<\&>} (a \texttt{<|>} b) &= a \\ a \texttt{<|>} (a \texttt{<\&>} b) &= a\end{aligned}$$

**Proof:**
First law:

$$\begin{aligned}&[\![a \texttt{<\&>} (a \texttt{<|>} b)]\!] \\ =&[\![a]\!] \cap \bigcap\{Y \mid [\![a]\!] \cup [\![b]\!] \subseteq Y, Y \text{ lazy domain}\} \\ =&\bigcap\{[\![a]\!] \cap Y \mid [\![a]\!] \cup [\![b]\!] \subseteq Y, Y \text{ lazy domain}\} \\ =&\bigcap\{Y \mid [\![a]\!] \cap ([\![a]\!] \cup [\![b]\!]) \subseteq Y, Y \text{ lazy domain}\} \\ =&\bigcap\{Y \mid [\![a]\!] \subseteq Y, Y \text{ lazy domain}\} \\ =&[\![a]\!]\end{aligned}$$

Second law:

$$\begin{aligned}&[\![a \texttt{<|>} (a \texttt{<\&>} b)]\!]\\ =&\bigcap\{Y \mid [\![a]\!] \cup ([\![a]\!] \cap [\![b]\!]) \subseteq Y, Y \text{ lazy domain}\}\\ =&\bigcap\{Y \mid [\![a]\!] \subseteq Y, Y \text{ lazy domain}\}\\ =&[\![a]\!]\end{aligned}$$

□

The two binary combinators are related via distributive laws:

LEMMA 13 (Distributive laws).

$$a \texttt{<|>} (b \texttt{<\&>} c) = (a \texttt{<|>} b) \texttt{<\&>} (a \texttt{<|>} c)$$
$$a \texttt{<\&>} (b \texttt{<|>} c) = (a \texttt{<\&>} b) \texttt{<|>} (a \texttt{<\&>} c)$$

**Proof:**
First law:

$$\begin{aligned}&[\![a \texttt{<|>} (b \texttt{<\&>} c)]\!]\\ =&\bigcap\{Y \mid [\![a]\!] \cup ([\![b]\!] \cap [\![c]\!]) \subseteq Y, Y \text{ lazy domain}\}\\ =&\bigcap\{Y \mid ([\![a]\!] \cup [\![b]\!]) \cap ([\![a]\!] \cup [\![c]\!]) \subseteq Y, Y \text{ lazy domain}\}\\ =&\bigcap\{Y \mid [\![a]\!] \cup [\![b]\!] \subseteq Y, [\![a]\!] \cup [\![c]\!] \subseteq Y, Y \text{ lazy domain}\}\\ =&\ \bigcap\{Y \mid [\![a]\!] \cup [\![b]\!] \subseteq Y, Y \text{ lazy domain}\}\\ &\cap \bigcap\{Y \mid [\![a]\!] \cup [\![c]\!] \subseteq Y, Y \text{ lazy domain}\}\\ =&[\![(a \texttt{<|>} b) \texttt{<\&>} (a \texttt{<|>} c)]\!]\end{aligned}$$

The second distributive law is known to be equivalent to the first. □

COROLLARY 14. *Lazy assertions form a bounded distributive lattice with meet* `<&>`*, join* `<|>`*, least element* `pNone` *and greatest element* `pAny`*. The ordering is the subset-relationship on acceptance sets.*

Thus all laws of bounded distributive lattices hold for lazy assertions, for example the idempotency laws:

$$a \texttt{<\&>} a = a$$
$$a \texttt{<|>} a = a$$

### 5.5 Negation

We cannot define any negation for lazy assertions that would turn them into a Boolean algebra. Let $a$ be an assertion with $[\![a]\!] = \{\bot, (\bot, \bot)\}$. From the first complement law of Boolean algebra, $a \texttt{<\&>} \neg a = \texttt{pNone}$, follows that $[\![a]\!] \cap [\![\neg a]\!] = \{\bot\}$. Laziness requires that $[\![\neg a]\!]$ is a lower set. So $[\![\neg a]\!] = \{\bot\}$. Hence $[\![a \texttt{<|>} \neg a]\!] = [\![a]\!]$. This contradicts the second complement law of Boolean algebra $a \texttt{<|>} \neg a = \texttt{pAny}$.

We will see that we can still define a useful, weaker variation of negation for constructor terms.

## 6. Implementing Assertions

In the previous section we developed a general semantics for lazy assertions for any domain. We have not yet said much about any concrete acceptance sets $[\![a]\!]$. For that we have to consider the concrete domains of specific types. We currently only have `pNone`, pAny and the combinators `<&>` and `<|>`, which on their own make for a rather boring set of assertions. Every domain needs its specific set of additional assertions and that is where we have design choices. Also, our semantic definitions are non-constructive. They just define acceptance sets, not algorithms for evaluating assertions. In particular our definition of `<|>` in terms of an intersection of a usually infinite set is hardly a good basis for an implementation.

### 6.1 Primitive Data Types

Every programming language includes some primitive data types such as number and characters that are flat domains, that is, $v \sqsubset w$ implies $v = \bot$. If a value of such a primitive data type is demanded in the computation, the whole atomic value is demanded. Hence like for strict assertions we can use predicates, Boolean-valued functions $\phi$ of our programming language, as assertions. This choice gives us maximal expressibility. We can for example define an assertion requiring a number to be prime.

We define the acceptance set of a predicate assertion for flat domains as

$$[\![\phi]\!] := \{\bot\} \cup \{v \mid \phi\, v\}$$

We have to explicitly include $\bot$ to ensure that the acceptance set is lower, as we may have $\phi\ \bot = \bot$ or $\phi\ \bot = \texttt{False}$. Predicate assertions meet our revised axioms: $[\![\phi]\!]$ is lower and $[\![\phi]\!]_v$ is directed for any $v$, because the domain is flat and these sets include $\bot$.

From the definition of the predicate acceptance set and our general assertion application definition we can derive an implementation of predicate assertion application:

$$\begin{aligned}\langle\phi\rangle\, v &= \bigsqcup \downarrow\{v\} \cap [\![\phi]\!]\\ &= \bigsqcup \{\bot, v\} \cap (\{\bot\} \cup \{w \mid \phi\, w\})\\ &= \bigsqcup \{\bot\} \cup (\texttt{if } \phi\, v \texttt{ then } \{v\} \texttt{ else } \{\bot\})\\ &= \texttt{if } \phi\, v \texttt{ then } v \texttt{ else } \bot\end{aligned}$$

The implementation works also for $v = \bot$, and no matter whether $\phi\bot = \bot$ or not.

Choosing $[\![\phi]\!] := \{\bot\} \cup \{v \mid \phi\, v \neq \texttt{False}\}$ looks like a good alternative definition for predicate assertions; it would also include all values $v$ with $\phi\, v = \bot$. However, if we try to derive the assertion application, then we find it un-implementable.

The least and greatest predicate assertions are

$$\texttt{pNone} := \lambda x.\texttt{False}$$
$$\texttt{pAny} := \lambda x.\texttt{True}$$

and the canonical definitions for conjunction and disjunction

$$\phi \texttt{<\&>} \psi = \lambda x.\phi\, x \wedge \psi\, x$$
$$\phi \texttt{<|>} \psi = \lambda x.\phi\, x \vee \psi\, x$$

do indeed meet our previous definitions:

$$\begin{aligned}[\![\phi \texttt{<\&>} \psi]\!] &= [\![\phi]\!] \cap [\![\psi]\!]\\ &= \{v \mid \phi\, v \wedge \psi\, v\}\\ [\![\phi \texttt{<|>} \psi]\!] &= \bigcap\{X \mid [\![\phi]\!] \cup [\![\psi]\!] \subseteq X, X \text{ lazy domain}\}\\ &= \bigcap\{X \mid [\![\phi]\!] \cup [\![\psi]\!] \subseteq X\}\\ &= [\![\phi]\!] \cup [\![\psi]\!]\\ &= \{v \mid \phi\, v \vee \psi\, v\}\end{aligned}$$

The latter proof works, because in flat domains any set $X$ including $\bot$ is lower, least upper bounds are trivial, and $\downarrow\{v\} \cap X$ is always directed.

Although negation is not generally available for lazy assertions, we could define it for flat domains:

$$\neg\phi := \lambda x.\neg(\phi\, x)$$

Its semantics meets our revised axioms, but it is unclear whether this combinator is useful in practise, as the programmer can manipulate the predicate directly.

### 6.2 Algebraic Data Types

Our main focus are data types whose values are constructor terms, such as the type `Form` for representing propositional formulae. Data constructors are symbols with an arity, including arity 0, and terms are built from full applications of a finite number of different data constructors to terms, and the value $\bot$. The domain is not flat. Often it even allows for infinite values (least upper bounds of a countable directed set of finite, partial approximations). The domain is usually not directed.

Like in our introductory example, we use constructors as patterns to describe assertions. A pattern assertion is built by applying a constructor to as many assertions as the arity of the constructor specifies. Thus constructor terms are a subset of constructor assertions, but together with other assertion combinators the assertion language becomes richer. Let $C$ be a data constructor of the programming language and $a_1, \ldots, a_n$ lazy assertions for its argument domains.

$$[\![C\ a_1 \ldots a_n]\!] := \{\bot\} \cup \{C\ v_1 \ldots v_n \mid v_1 \in [\![a_1]\!] \ldots v_n \in [\![a_n]\!]\}$$

This definition meets our revised axioms of lazy assertions: Because $a_1, \ldots, a_n$ are lazy assertions, $[\![a_1]\!], \ldots, [\![a_n]\!]$ are lower and $[\![a_1]\!]_{v_1}, \ldots, [\![a_n]\!]_{v_n}$ are directed for any $v_1, \ldots v_n$ of the respective domains. Hence $[\![C\ a_1 \ldots a_n]\!]$ is lower and $[\![C\ a_1 \ldots a_n]\!]_v$ is directed for all $v$.

Now we look for constructive definitions of our two combinators `<&>` and `<|>`. For `<&>` we easily find the following two laws:

LEMMA 15 (Conjunction of constructor assertions).

$$(C\ a_1 \ldots a_n)\ \texttt{<\&>}\ (C\ b_1 \ldots b_n) = C(a_1\ \texttt{<\&>}\ b_1) \ldots (a_n\ \texttt{<\&>}\ b_n)$$
$$(C\ a_1 \ldots a_n)\ \texttt{<\&>}\ (C'\ b_1 \ldots b_n) = \texttt{pNone} \qquad \text{if } C \neq C'$$

**Proof:**
First law:

$$\begin{aligned}
&[\![(C\ a_1 \ldots a_n)\ \texttt{<\&>}\ (C\ b_1 \ldots b_n)]\!]\\
=&[\![C\ a_1 \ldots a_n)]\!] \cap [\![C\ b_1 \ldots b_n]\!]\\
=&(\{\bot\} \cup \{C\ v_1 \ldots v_n \mid v_1 \in [\![a_1]\!], \ldots, v_n \in [\![a_n]\!]\}) \cap\\
&(\{\bot\} \cup \{C\ v_1 \ldots v_n \mid v_1 \in [\![b_1]\!], \ldots, v_n \in [\![b_n]\!]\})\\
=&(\{\bot\} \cup \{C\ v_1 \ldots v_n \mid v_1 \in [\![a_1]\!] \cap [\![b_1]\!], \ldots, v_n \in [\![a_n]\!] \cap [\![b_n]\!]\})\\
=&[\![C(a_1\ \texttt{<\&>}\ b_1) \ldots (a_n\ \texttt{<\&>}\ b_n)]\!]
\end{aligned}$$

Second law:

$$\begin{aligned}
&[\![(C\ a_1 \ldots a_n)\ \texttt{<\&>}\ (C'\ b_1 \ldots b_n)]\!]\\
=&(\{\bot\} \cup \{C\ v_1 \ldots v_n \mid v_1 \in [\![a_1]\!], \ldots, v_n \in [\![a_n]\!]\}) \cap\\
&(\{\bot\} \cup \{C'\ v_1 \ldots v_n \mid v_1 \in [\![b_1]\!], \ldots, v_n \in [\![b_n]\!]\})\\
=&\{\bot\}\\
=&[\![\texttt{pNone}]\!]
\end{aligned}$$

□

Our general definition of `<|>` is more complicated than our definition of `<&>`. So let us look at some of its details to see what happens for constructor terms. Consider the set

$$S := \{\bot\} \cup \{C\ v_1\ v_2 \mid (v_1 \in [\![a_1]\!] \wedge v_2 \in [\![a_2]\!]) \vee (v_1 \in [\![b_1]\!] \wedge v_2 \in [\![b_2]\!])\}$$

In general $\downarrow\{v\} \cap S$ is not directed for all values $v$. Consider for example the elements $C\ v_1\ \bot$ and $C\ \bot\ v_2$ of $S$, with $v_1 \in [\![a_1]\!]$ and $v_2 \in [\![b_2]\!]$. Their least upper bound is $C\ v_1\ v_2$, which is not in $S$ by definition. So any (lower) $Y$ with $S \subseteq Y$ and $\downarrow\{v\} \cap Y$ directed must include these least upper bounds. In other words, any such $Y$ must be a superset of

$$\{\bot\} \cup \{C\ v_1\ v_2 \mid (v_1 \in [\![a_1]\!] \vee v_1 \in [\![b_1]\!]) \wedge (v_2 \in [\![a_2]\!] \vee v_2 \in [\![b_2]\!])\}$$

Hence the following holds:

$$\begin{aligned}
&[\![(C\ a_1 \ldots a_n)\ \texttt{<|>}\ (C\ b_1 \ldots b_n)]\!]\\
=&\bigcap\{Y \mid \{\bot\} \cup \{C\ \overline{v} \mid (v_1 \in [\![a_1]\!] \wedge \ldots \wedge v_n \in [\![a_n]\!]) \vee\\
&\qquad (v_1 \in [\![b_1]\!] \wedge \ldots \wedge v_n \in [\![b_n]\!])\} \subseteq Y,\\
&\quad Y \text{ lazy domain}\}\\
=&\bigcap\{Y \mid \{\bot\} \cup \{C\ \overline{v} \mid (v_1 \in [\![a_1]\!] \vee v_1 \in [\![b_1]\!]) \wedge \ldots \wedge\\
&\qquad (v_n \in [\![a_n]\!] \vee v_n \in [\![b_n]\!])\} \subseteq Y,\\
&\quad Y \text{ lazy domain}\}\\
=&\{\bot\} \cup \{C\ \overline{v} \mid v_1 \in \bigcap\{Y_1 \mid [\![a_1]\!] \cup [\![b_1]\!] \subseteq Y_1,\\
&\qquad Y_1 \text{ lazy domain}\}, \ldots,\\
&\qquad v_n \in \bigcap\{Y_n \mid [\![a_n]\!] \cup [\![b_n]\!] \subseteq Y_n,\\
&\qquad Y_n \text{ lazy domain}\}\}\\
=&\{\bot\} \cup \{C\ \overline{v} \mid v_1 \in [\![a_1\ \texttt{<|>}\ b_1]\!], \ldots, v_n \in [\![a_n\ \texttt{<|>}\ b_n]\!]\}\\
=&[\![C\ (a_1\ \texttt{<|>}\ b_1) \ldots (a_n\ \texttt{<|>}\ b_n)]\!]
\end{aligned}$$

So we find that for constructor terms the first law for `<|>` is surprisingly similar to that for `<&>`:

LEMMA 16 (Disjunction of constructor assertions).

$$(C\ a_1 \ldots a_n)\ \texttt{<|>}\ (C\ b_1 \ldots b_n) = C\ (a_1\ \texttt{<|>}\ b_1) \ldots (a_n\ \texttt{<|>}\ b_n)$$

For different constructors no simple law exists, but we note that if $C \neq C'$, then $Y := [\![C\ a_1 \ldots a_n]\!] \cup [\![C'\ b_1 \ldots b_n]\!]$ is lower. Least upper bounds are preserved and $\downarrow\{v\} \cap Y$ is directed for any value $v$, because $[\![C\ a_1 \ldots a_n]\!] \cap [\![C'\ b_1 \ldots b_n]\!] = \{\bot\}$. Hence

$$\begin{aligned}
&[\![(C\ a_1 \ldots a_n)\ \texttt{<|>}\ (C'\ b_1 \ldots b_n)]\!]\\
=&\bigcap\{Y \mid [\![C\ a_1 \ldots a_n]\!] \cup [\![C'\ b_1 \ldots b_n]\!] \subseteq Y,\\
&\quad Y \text{ lazy domain}\}\\
=&[\![C\ a_1 \ldots a_n]\!] \cup [\![C'\ b_1 \ldots b_n]\!]
\end{aligned}$$

Altogether we see that although the general definition of `<|>` in terms of an infinite intersection is not simple, its incarnation for constructors is easy to use. In our introductory examples and most applications the two assertions combined by `<|>` have either disjoint acceptance sets, in which case we get a simple union of acceptance sets, or they have the same acceptance sets, in which case idempotency gives us as expected $a\ \texttt{<|>}\ a = a$. For the general case of a partial overlap we have Lemma 16.

Lazy assertions form a bounded distributive lattice, there is no general negation. However, in our initial example we noted that it is very useful to be able to express that an assertion does not accept certain top-level constructors. Hence we introduce a specific subtraction combinator on assertions that removes a constructor. Let $a$ be a lazy assertion and $c$ be a constructor of its domain.

$$[\![a\texttt{<\textbackslash>}C]\!] := [\![a]\!] \backslash \{C\ v_1 \ldots v_n \mid v_1, \ldots, v_n \text{ any domain values}\}$$

This definition meets our revised axioms, $[\![a\texttt{<\textbackslash>}C]\!]$ is lower and $[\![a\texttt{<\textbackslash>}C]\!]_v$ directed for any $v$ if $a$ meets the axioms.

The combinator meets distribution laws:

$$(a \texttt{<\&>} b) \texttt{<\>} C = (a\texttt{<\>}C) \texttt{<\&>} (b\texttt{<\>}C)$$
$$(a \texttt{<|>} b) \texttt{<\>} C = (a\texttt{<\>}C) \texttt{<|>} (b\texttt{<\>}C)$$

More important for an implementation is that the following laws trivially hold:

LEMMA 17 (Subtraction for constructor assertions).

$$C\ \overline{a} \texttt{<\>} C = \texttt{pNone}$$
$$C\ \overline{a} \texttt{<\>} C' = C\ \overline{a} \qquad \text{if } C \neq C'$$

### 6.3 Implementation

We have not yet actually given a complete implementation of the assertion combinators for constructor terms. However, the laws of our lemmas and the general distributive lattice laws are sufficient to derive an efficient implementation.

The laws allow us to represent any constructor assertion of any algebraic data type as a finite disjunction

$$C_1\ \overline{a}_1 \texttt{<|>} C_2\ \overline{a}_2 \texttt{<|>} \ \ldots\ \texttt{<|>} C_m\ \overline{a}_m$$

where $\{C_1, \ldots, C_m\}$ is a subset of the constructors of the algebraic data type.

Clearly any simple constructor assertion $C\ \overline{a}$ is already in that form. The least assertion `pNone` is represented as the empty disjunction and

$$\texttt{pAny} = C_1\ \overline{\texttt{pAny}} \texttt{<|>} \ \ldots\ \texttt{<|>} C_n\ \overline{\texttt{pAny}}$$

The definition of assertion application for our representation is straightforward:

$$\langle C_1\ \overline{a}_1 \texttt{<|>} \ \ldots\ \texttt{<|>} C_m\ \overline{a}_m\rangle\ (C\ \overline{v}) = \begin{cases} C\ (\langle \overline{a}_j \rangle\ \overline{v}) & \text{if } C = C_j \\ \bot & \text{otherwise} \end{cases}$$
$$\langle C_1 \overline{a}_1 \texttt{<|>} \ \ldots\ \texttt{<|>} C_m\ \overline{a}_m\rangle\ \bot = \bot$$

Here $\langle \overline{a}_j \rangle\ \overline{v}$ is an abbreviation for component-wise assertion application.

Using our laws for constructor assertions we can define conjunction and disjunction on our representation:

$$(C_{i_1}\ \overline{a}_{i_1} \texttt{<|>} \ldots \texttt{<|>} C_{i_m}\ \overline{a}_{i_m}) \texttt{<\&>} (C_{j_1}\ \overline{b}_{j_1} \texttt{<|>} \ldots \texttt{<|>} C_{j_l}\ \overline{b}_{j_l})$$
$$= C_{k_1}\ (\overline{a}_{k_1} \texttt{<\&>} \overline{b}_{k_1}) \texttt{<|>} \ldots \texttt{<|>} C_{k_o}\ (\overline{a}_{k_o} \texttt{<\&>} \overline{b}_{k_o})$$

where

$$\{k_1, \ldots, k_o\} = \{i_1, \ldots, i_m\} \cap \{j_1, \ldots, j_l\}$$

and

$$(C_{i_1}\ \overline{a}_{i_1} \texttt{<|>} \ldots \texttt{<|>} C_{i_m}\ \overline{a}_{i_m}) \texttt{<|>} (C_{j_1}\ \overline{b}_{j_1} \texttt{<|>} \ldots \texttt{<|>} C_{j_l}\ \overline{b}_{j_l})$$
$$= C_{k_1}\ \overline{z}_{k_1} \texttt{<|>} \ldots \texttt{<|>} C_{k_o}\ \overline{z}_{k_o}$$

where

$$\{k_1, \ldots, k_o\} = \{i_1, \ldots, i_m\} \cup \{j_1, \ldots, j_l\}$$
$$z_{k_s} = \begin{cases} \overline{a}_{k_s} \texttt{<|>} \overline{b}_{k_s} & \text{if } k_s \in \{i_1, \ldots, i_m\} \cap \{j_1, \ldots, j_l\} \\ \overline{a}_{k_s} & \text{if } k_s \in \{i_1, \ldots, i_m\} \backslash \{j_1, \ldots, j_l\} \\ \overline{b}_{k_s} & \text{if } k_s \in \{j_1, \ldots, j_l\} \backslash \{i_1, \ldots, i_m\} \end{cases}$$

In these equations $\overline{a} \texttt{<|>} \overline{b}$ and $\overline{a} \texttt{<\&>} \overline{b}$ are abbreviations for the component-wise combinations.

The definition of `<\>` for this representation is simple: it just removes one of the elements of the disjunction.

This implementation is efficient, because the disjunctive representation of an assertion is computed only once, afterwards assertion application for constructor terms requires linear time in the size of the checked data structure. By introducing an additional representation of `pAny` we can also avoid doing any checking work on a data structure that is accepted anyway. The efficiency of the function assertions for primitive types fully depends on the efficiency of the predicates that the programmer chooses.

A prototype implementation using this representation, including predicate assertions for primitive types, exists in Haskell and has been used in several examples, including the propositional formulae of Section 2. Associated types [3, 4] enable providing a single class with functions `assert`, `<|>`, etc. with different implementations for each type.

### 6.4 What about Function Types?

The domain of functions over a domain $D$ and codomain $D'$ is the set of continuous functions $[D \to D']$ from the domain to the codomain.

We follow previous work on assertions for strict functional languages [10] and define a function assertion as a tuple $a \mapsto b$ of an assertion $a$ for the domain, the pre-condition, and an assertion $b$ for the codomain, the post-condition. We can define the standard function assertion application [10] simply by composing existing assertion applications:

$$\langle a \mapsto b\rangle\ \delta = \lambda x.\langle b\rangle\ (\delta\ (\langle a\rangle\ x))$$

Thus the assertion correctly checks both the pre- and the post-condition. Function arguments and results are checked lazily, only as far as they are demanded by the context. We can compose function assertions to build assertions for higher-order functions.

However, our definition of the acceptance set yields

$$[\![a \mapsto b]\!] = \{\delta \in [D \to D'] \mid \langle b\rangle \circ \delta \circ \langle a\rangle = \delta\}$$

which is not a lazy domain! It is not even a lower set.

Instead, comparison with the eager quotient model [2] suggests

$$[\![a \mapsto b]\!] = \{\delta \in [D \to D'] \mid \forall v \in [\![a]\!].\ \delta\ v \in [\![b]\!]\}$$

and comparison with the eager projection model [9] suggests

$$[\![a \mapsto b]\!] = \{\delta \in [D \to D'] \mid \forall v \in D.\ \delta\ v \in [\![b]\!]\}$$

Both these sets are lazy domains. However, both acceptance sets describe only half of the assertion application. Neither of them expresses that if the function argument is not within the acceptance set $[\![a]\!]$, then an exception should be raised, because the pre-condition is violated.

Function assertions are different from first-order assertions in that they describe properties of both the assertion argument, the function, and the context of the assertion application (which provides the argument for the function).

Findler and Blume solved this problem in their projection model [9] by interpreting an eager function assertion as a *pair* of projections. One projection restricts the assertion argument, the other restricts the context. For first-order assertions the context projection is just the identity function. Similarly we could use either pairs of lazy projections (cf. Theorem 4) or introduce a second acceptance set restricting the context. We believe that the definitions and lemmas of previous sections can be transferred.

There is still more potential for lazy function assertions: For eager functional languages there exist dependent function assertions, where the assertion of the function result depends on the argument passed to the function. For algebraic data types lazy assertions cannot relate different subterms of constructor terms, because the subterms can be demanded in any order. Relating argument and result of a function is different, because only after the result of a function has been demanded, the argument may be demanded. Hence it should be possible to express non-strictness properties such that only when a certain part of the result of a function is demanded, the function demands a certain part of its argument.

## 7. Discussion of Related Work

There is an extensive literature on assertions and contracts for non-functional programming languages. A contract combines several assertions. A contract consists of a pre-condition that a function caller or client has to meet and a post-condition that the function or server promises in return [19]. When the contract is violated, one of the parties is blamed. In this paper we discussed only assertions. Contracts with blaming can be built on top of them.

Assertion-based contracts became popular in eager functional programming languages with Findler and Felleisen's seminal paper on contracts for higher-order functions [10]. All interesting properties of functional values, which are passed around by higher-order functions, are undecidable; it is impossible to check a function contract for all argument-result-pairs. However, Findler and Felleisen realised that it is sufficient to check both the pre- and the post-condition of a functional value only when this function is applied. The resulting contract system is sound.

This "lazy" checking of contracts for functional values in eager languages provides another argument for using lazy assertions in lazy languages: A lazy data type `T` is isomorphic to the function type `Unit -> T` in an eager language. Demanding a value of the lazy type `T` corresponds to applying the function of the eager type `Unit -> T`. Hence because an assertion for a function is checked only when the function is applied to an argument, an assertion for a lazy value should only be checked when the value is demanded.

Other contributions of Findler and Felleisen's paper, a system for correctly attributing blame in case of contract violation and an implementation as an extension of a Scheme runtime system are not relevant for this paper, as the implementation approach is different. The work also provides dependent function contracts, where the assertion for the codomain can use the actual argument value.

Hinze et al. [17] transferred contracts for higher-order functions to Haskell, implementing it as a library instead of modifying the runtime system. Thus it will provide a good model for building contracts and blame assignment on top of our assertions. However, their semantics is a seemingly random mixture of eager and lazy assertion evaluation.

Lazy assertions were first discussed and several implementations presented in 2004 [7]. That paper makes the point that while eager assertions must be `True`, lazy assertions must not be `False`, which lead us here to require that acceptance sets are lower. The paper uses predicates on values of all types and hence, despite some technical tricks using concurrency, the assertions are lazy but neither trustworthy nor prompt. The paper itself gives examples of where assertion violations are noticed too late. This problem was later rectified [5, 6]. 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 trustworthy implementation [6] uses patterns similar to those in this paper to express assertions over algebraic data types. However, a non-deterministic implementation of disjunction leads to the semantic problems described in the introduction. Later [5] provided a more user-friendly language for expressing assertions and improved the internal structure of the implementation, but the implementation principles were identical and hence the non-deterministic disjunction remained. The axioms of our Definition 2 are inspired by this previous work, but there was no formal semantics and the implementation derived here is completely different.

Degen et al. [8] classify all existing assertion systems for Haskell as eager (straight translation of [10]), semi-eager [17] and lazy [5–7]. They check whether the systems meet their four desirable properties of meaning reflection, meaning preservation, faithfulness and idempotency. Sadly, none of the assertion systems meets all four properties. In particular, they note that lazy assertions are not faithful. Hence, although eager assertions are not meaning preserving for lazy languages, they prefer them and conclude with the slogan "Faithfulness is better than laziness". The lazy assertions presented in this paper are idempotent and also meaning reflecting and meaning preserving, as far as the informal definitions of the latter properties allow such a statement. To demonstrate that lazy assertions are not faithful, Degen et al. consider the expression

```
assert (pred (==0) |-> pAny) (\x -> 42) 5
```

Evaluating it yields `42` in all lazy assertion systems, including the one presented in this paper. However, faithfulness would require evaluation to yield an exception, complaining that `5` is unequal `0`. Faithfulness is similar to trustworthiness, but stricter for function assertions. In this paper, the semantic value passed as argument to the function `(\x -> 42)` is the exception $\bot$. So the function gets an argument within the acceptance set and hence the assertion is trustworthy. The expression above is equivalent to

```
(\x -> 42) (assert (pred (==0)) 5)
```

It is important to know what an assertion guarantees and what not. The semantics defined in this paper provides an answer. Faithfulness seems to be unnecessarily too strong a requirement. Faithfulness conflicts with laziness, because acceptance sets are lower. Trustworthiness suffices.

Blume et al. proposed and studied several semantic models of Findler and Felleisen's higher-order contracts [2, 9, 11]. These semantic models are based on operational semantics. The quotient model [2] equates a contract with the set of terms satisfying the contract. The property of safety, that is, whether a term contains a blame exception that can be raised by a context, plays a major role. The projection model [9, 11] interprets a contract as a pair of projections and defines a suitable ordering on these pairs of projections. 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 primitive types. Instead the papers focus on higher-order functions and correct blame assignment. They are more complex than our simple semantics of lazy assertions. To define a semantics for lazy assertions incorporating higher-order functions we probably have to combine ideas from both worlds.

For lazy functional logic languages Hanus [16] proposed to allow the user to choose for each assertion whether it should be eager or lazy, depending on whether meaning preservation or faithfulness are essential. Assertion checking can even be delayed until a point in the program execution, e.g. just before an important output action. The prototypical implementation combines ideas from eager [10] and early lazy assertions [7], taking advantage of some logical language features. The semantics is not studied, but as functional logic languages permit non-deterministic functions, older — more expressive — lazy assertions may still fit within the semantics of the language [18]. It would be interesting to determine whether that is the case, or the fact that the non-determinism of an assertion depends on its context would still break the language semantics.

The original idea of assertions is that they are checked at runtime. However, there exist several proposals for doing at least part of the checking statically, like type checking [12, 22]. The main challenge is to handle the intrinsic undecidability of assertions. The proposal for Haskell [22] is not lazy in the sense of this paper; acceptance sets are not lower. That is fine; we require our laziness only because our assertions are checked at runtime.

Work on improving the implementation of eager higher-order contracts for Scheme are ongoing [13]. These ideas are closely linked to the eager semantics of Scheme.

Our semantics uses basic domain theory. Acceptance sets are *inclusive* subsets as they are often used to interpret types [14]. As-

sertions are *finitary* projections, widely used for defining domains [14, 15]. Our assertion sets with respect to a given value, $[\![a]\!]_v$, are *normal* in $[\![a]\!]$ [15]. The definition of $[\![a]\!]_v$ also reminds of *bases of domains* [1]. Still, the specific combination of axioms for lazy assertions made it hard to reuse many standard theorems of domain theory.

## 8. Conclusions

We have defined a semantics for lazy assertions. Our semantics is simple; it is based on the acceptance set $[\![a]\!]$ of an assertion $a$ as the subset of the domain that contains all accepted values. Our axioms are few and clarify what lazy assertions are. The axioms yield a beautiful algebra of assertions, a bounded distributive lattice. It turns out that for primitive and algebraic data types the axioms leave little freedom in choosing suitable assertion combinators. From the semantics we derived an efficient prototype implementation of lazy assertions for Haskell. The implementation is just a library, thus requiring no specific runtime system support and guaranteeing to preserve the language semantics.

Associated type classes provide a simple interface to lazy assertions, but in the future we have to find a suitable generic programming framework to enable us to define lazy assertions once for all types, without any tedious repetition for pattern combinators and even combinators such as conjunction for every type.

Previous lazy assertions, which break standard semantic equivalences, can express properties such as a list being ordered. However, checking such an assertion can take time that is exponential in the length of the list. Hence there is an advantage in that the lazy assertions presented here are limited to properties of algebraic data types that can be checked in linear time. It is a limitation that simple equality checks like for the Boolean tuple in the introduction cannot be expressed. However, such assertions can only be violated when data structures are fully evaluated, which may happen less frequently in programs that work with many partial data structures (cf. the Pasta interpreter application in [7]) and thus may be less useful. Lazy assertions for algebraic data types are, however, quite different from those for eager functional languages. Existing functions and predicates cannot be reused but instead the pattern matching assertion combinators have to be used; the definition of lazy assertions is similar to the definition of algebraic data types. Thus lazy assertions for algebraic data types are less like eager assertions but more like subtypes expressing context-free properties, underlined by the fact that acceptance sets are subdomains. In principle, variants of the original algebraic data type could be used instead. For example, each intermediate abstract syntax tree in a multi-pass compiler could be specified as a separate algebraic data type, but subtypes are far more convenient to write and read.

The aim of this paper was to study *lazy* assertions. Laziness mainly influences the semantics of algebraic data types, hence these were the focus of our studies. It is reassuring that predicate-based assertions for primitive types seamlessly fit into this framework. Lazy assertions for functions still require further work. We are confident that the established implementation of non-dependent function assertions for eager functional languages can also be fitted within our lazy semantic framework. More exciting but yet unanswered is the prospect of developing a form of dependent function assertions. These function assertions will go far beyond standard subtypes.

## Acknowledgments

I thank Colin Runciman and Frank Huch for many earlier discussions about lazy assertions.

## References

[1] S. Abramsky and A. Jung. Domain theory. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 3, pages 1–168. Clarendon Press, 1994.

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

[3] M. M. T. Chakravarty, G. Keller, and S. P. Jones. Associated type synonyms. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 241–253. ACM, 2005.

[4] M. M. T. Chakravarty, G. Keller, S. P. Jones, and S. Marlow. Associated types with class. In *POPL '05: Symposium on Principles of programming languages*, pages 1–13. ACM, 2005.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] C. Flanagan. Hybrid type checking. In *POPL '06: Symposium on Principles of programming languages*, pages 245–256. ACM, 2006.

[13] M. Greenberg, B. C. Pierce, and S. Weirich. Contracts made manifest. In *POPL '10: Symposium on Principles of programming languages*, pages 353–364. ACM, 2010.

[14] C. A. Gunter. *Semantics of programming languages: structures and techniques*. MIT Press, 1992.

[15] C. A. Gunter and D. S. Scott. Semantic domains. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 633–674. MIT Press, 1990.

[16] M. Hanus. Lazy and faithful assertions for functional logic programs. In *Proc. of the 19th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2010)*, pages 50–64. Universidad Politécnica de Madrid, 2010.

[17] 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.

[18] F. J. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. A simple rewrite notion for call-time choice semantics. In *PPDP '07*, pages 197–208, 2007.

[19] B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.

[20] D. L. Parnas. A technique for software module specification with examples. *Commun. ACM*, 15(5):330–336, 1972.

[21] D. S. Rosenblum. A practical approach to programming with assertions. *IEEE Trans. Softw. Eng.*, 21(1):19–31, 1995.

[22] D. N. Xu, S. Peyton Jones, and K. Claessen. Static contract checking for Haskell. In *POPL '09: Symposium on Principles of programming languages*, pages 41–52. ACM, 2009.