# A Semantics for Lazy Assertions

Olaf Chitil

University of Kent

25th January 2011

# Assertions in Functional Languages

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

Assertion application is a partial identity.

```
assert :: Assertion t -> t -> t
nats :: Num t => Assertion [t]
```

Note: Contract = Assertion + Blaming

# Lazy Assertions ...

... work with non-strict functions and infinite data structures.

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

Need to consider partial values:

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

Any approximation of an acceptable value has to be accepted!

# The Problem

```
let x = assert equal (True,False)
in (fst x, snd x)
```
$\leadsto$ exception

but

```
(fst (assert equal (True,False)),
 snd (assert equal (True,False)))
```
$\leadsto$
```
(True,
 False)
```

Because
$$\begin{array}{lcl} \texttt{(True,}\bot\texttt{)} & \sqsubseteq & \texttt{(True,True)} \\ \texttt{(}\bot\texttt{,False)} & \sqsubseteq & \texttt{(False,False)} \end{array}$$
have to be accepted.

# The Problem

```
let x = assert equal (True,False)  ⇝  (True, error "...")   or
in (fst x, snd x)                      (error "...", False)
```

but

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

Because $\begin{array}{lcl} \texttt{(True,}\bot\texttt{)} & \sqsubseteq & \texttt{(True,True)} \\ \texttt{(}\bot\texttt{,False)} & \sqsubseteq & \texttt{(False,False)} \end{array}$ have to be accepted.

Hence: First define semantics, then derive an implementation.

# Resulting Lazy Assertions: List of natural numbers

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

aList :: Assert t -> Assert [t]
aList a = aNil <|> aCons a (aList a)
```

# Resulting Lazy Assertions: Minimal list length

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

initAverage :: [Int] -> Int
initAverage = assert (lengthAtLeast 5 |-> aAny) initAverage'

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

# Resulting Lazy Assertions: Logic Example

Data type for formulae:

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

Assertions for conjunctive normal form:

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

conjNF = aAnd conjNF conjNF <|> disj
disj = aOr disj disj <|> lit
lit = aNot atom <|> atom
atom = aAtom aAny
```

Conjunctive normal form with left-associated operators:

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

# Axioms of Semantics

Write $\langle a \rangle : D \to D$ for semantics of `assert a`.
Domain $D$ is directed complete partial order with $\bot$.

**Definition**

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

**Definition**

$a$ is lazy assertion, if

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 $v$, (equivalent: $a$ is idempotent)
3. $\langle a \rangle$ is a partial identity, that is, $\langle a \rangle\, v \sqsubseteq v$ for any value $v$, and
4. $[\![a]\!]$ is a lower set.

# Assertions and Projections

**Definition**

A function $p : D \to D$ on a domain $D$ is a projection if it is

- continuous,
- idempotent, and
- a partial identity.

**Lemma**

$a$ is lazy assertion $\Leftrightarrow$ $\langle a \rangle$ is projection and its image is a lower set

(cf. Findler & Blume, FLOPS 2006)

# Looking for Alternative Axioms

## Definition

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

Theorem

$[\![a]\!]_v$ is an ideal (lower & directed)

$$\langle a \rangle\ v = \bigsqcup [\![a]\!]_v$$

# Alternative Axioms

## Definition

A set $A \subseteq D$ is a lazy domain if

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

### Lemma

If $a$ is a lazy assertion, then $[\![a]\!]$ is a lazy domain.

### Theorem

If $A$ is a lazy domain, then $a$ with

$$\langle a \rangle\, v := \bigsqcup [\![a]\!]_v$$

is a lazy assertion with $[\![a]\!] = A$.

# Assertion Combinators: Minimal & Maximal

**Definition**

$$[\![\texttt{aNone}]\!] := \{\bot\}$$
$$[\![\texttt{aAny}]\!] := D$$

Derived assertion applications

$$\langle\texttt{aNone}\rangle\; v = \bigsqcup [\![\texttt{aNone}]\!]_v = \bigsqcup \downarrow\!\{v\} \cap \{\bot\} = \bigsqcup\{\bot\} = \bot$$

$$\langle\texttt{aAny}\rangle\; v = \bigsqcup [\![\texttt{aAny}]\!]_v = \bigsqcup \downarrow\!\{v\} \cap D = \bigsqcup \downarrow\!\{v\} = v$$

# Assertion Combinators: Conjunction

**Definition**

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

Lemma Conjunction of assertions is commutative and associative and has the assertion `aAny` as neutral element.

Lemma (Conjunction equals two assertions)

$$\langle a\texttt{<\&>}b\rangle\ v = \langle a\rangle\ (\langle b\rangle\ v)$$

# Assertion Combinators: Disjunction

Not $[\![a \vee b]\!] := [\![a]\!] \cup [\![b]\!]$
because $[\![a \vee b]\!]_v = (\downarrow\{v\} \cap [\![a]\!]) \cup (\downarrow\{v\} \cap [\![b]\!])$ not directed.

**Definition**

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

Attention! $D = \{\bot, (\bot,\bot), (\texttt{True},\bot), (\texttt{False},\bot), \ldots, (\texttt{False},\texttt{False})\}$

$$[\![\texttt{fstTrue}]\!] = D \setminus \{(\texttt{False},\bot), (\texttt{False},\texttt{True}), (\texttt{False},\texttt{False})\}$$

$$[\![\texttt{fstTrue <|> sndTrue}]\!] = D$$

$$[\![\texttt{(fstTrue <\&> sndTrue) <|> (fstFalse <\&> sndFalse)}]\!] = D$$

Lemma Disjunction of assertions is commutative and associative and has the assertion `aNone` as neutral element.

# Bounded Distributive Lattice of Assertions

Lemma (Absorption laws)

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

Lemma (Distributive laws)

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

Theorem Lazy assertions form a bounded distributive lattice with meet `<&>`, join `<|>`, least element `aNone` and greatest element `aAny`. The ordering is the subset-relationship on acceptance sets.

Corollary (Idempotency laws)

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

# No Negation

Let $[\![a]\!] := \{\perp, (\perp, \perp)\}$

$a \,\texttt{<\&>}\, \neg a = \texttt{aNone}$ implies $[\![a]\!] \cap [\![\neg a]\!] = \{\perp\}$.

$[\![\neg a]\!]$ must be a lower set.

So $[\![\neg a]\!] = \{\perp\}$.

But then $[\![a \,\texttt{<|>}\, \neg a]\!] = [\![a]\!]$.

Contradiction to $a \,\texttt{<|>}\, \neg a = \texttt{aAny}$.

# Deriving an Implementation: Primitive Data Types

Flat domain, i.e., $v \sqsubset w$ implies $v = \bot$.

**Definition (Acceptance set of predicate assertion)**

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

Derive application of assertion predicate:

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

Note: $\{\bot\} \cup \{v \mid \phi\ v \neq \mathtt{False}\}$ as acceptance set is un-implementable.

# Primitive Data Types: Conjunction & Disjunction

Expected definitions:

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

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

Verify they work:

$$[\![\phi \texttt{<\&>} \psi]\!] = [\![\phi]\!] \cap [\![\psi]\!] = \{v \mid \phi\, v \wedge \psi\, v\}$$

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

Negation is possible:

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

# Deriving an Implementation: Algebraic Data Types

**Definition (Acceptance set for pattern assertion)**

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

**Lemma** (Conjunction of constructor assertions)

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

**Lemma** (Disjunction of constructor assertions)

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

Also if $C \neq C'$, then

$$[\![(C\ a_1 \dots a_n) \texttt{<|>} (C'\ b_1 \dots b_n)]\!] = [\![C\ a_1 \dots a_n]\!] \cup [\![C'\ b_1 \dots b_n]\!]$$

# Assertion Representation and Application

Representation of constructor assertion

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

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

Application of a constructor assertion

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

# Algebraic Data Types

## Conjunction

$$
\begin{aligned}
& (C_{i_1}\,\overline{a}_{i_1} <|> \ldots <|> C_{i_m}\,\overline{a}_{i_m}) <\&> (C_{j_1}\,\overline{b}_{j_1} <|> \ldots <|> C_{j_l}\,\overline{b}_{j_l}) \\
= \quad & C_{k_1}\,(\overline{a}_{k_1} <\&> \overline{b}_{k_1}) <|> \ldots <|> C_{k_o}\,(\overline{a}_{k_o} <\&> \overline{b}_{k_o})
\end{aligned}
$$

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

## Disjunction

$$
\begin{aligned}
& (C_{i_1}\,\overline{a}_{i_1} <|> \ldots <|> C_{i_m}\,\overline{a}_{i_m}) <|> (C_{j_1}\,\overline{b}_{j_1} <|> \ldots <|> C_{j_l}\,\overline{b}_{j_l}) \\
= \quad & C_{k_1}\,\overline{z}_{k_1} <|> \ldots <|> C_{k_o}\,\overline{z}_{k_o}
\end{aligned}
$$

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

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

# What about Function Types?

Function Assertion $a \mapsto b$

Standard definition of assertion application:

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

But

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

is not a lower set! However,

$$\{\delta \mid \forall v \in [\![ a ]\!].\ \delta\, v \in [\![ b ]\!]\}$$

is a lazy domain.

Need two acceptance sets, for argument and context.

(cf. Findler & Blume, FLOPS 2006)

# Last Slide

## Semantics

- Only few axioms: functional, trustworthy, partial identity, lower set.
- Acceptance sets $[\![a]\!]$ are lazy domains, subdomains.
- Algebra of assertions: bounded distributive lattice.

## Lazy Assertions

- Derived as library from semantics.
- Laziness restricts expressibility!
- Pattern assertions similar to algebraic data types; subtypes!
- Pattern assertions are efficient.

## Future

- Assertions to express non-strictness properties.
- (Dependent?) function assertion semantics.

# Example: Normalisation of <&>

Formula in conjunctive normal form with left-associated binary operators:

```
conjNF = aAnd conjNF conjNF <|> aOr disj disj <|> aNot atom <|>
         aAtom aAny
```

```
left = aImp left noImp <|> aAnd left noAnd <|> aOr left noOr <|>
       aNot left <|> aAtom aAny
```

Combined:

```
leftConjNF = conjNF <&> left
           = aAnd (conjNF <&> left) (conjNF <&> noAnd)  <|>
             aOr (disj <&> left) (disj <&> noOr)  <|>
             aNot (atom <&> left)  <|>
             aAtom (aAny <&> aAny)
```