From Assertions to Contracts

- specify & dynamically check properties
- more expressive than static types, less effort than verification
- testing with real values

In functional languages assertion application is a partial identity.

```text
assert (prop (>= 0)) 42  \implies  42
assert (prop (>= 0)) (-2)  \implies  exception
```
Contracts

Systematic use of assertions as contract between a server and a client, separating their responsibilities.

Function contract:
- pre-condition has to be met by caller of the function
- post-condition has to be met by function itself

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

clausalNF :: Formula -> [[[Formula]]]

cClausalNF = assert (conjNF & right =>> list (list lit)) clausalNF
```

For Scheme: [Findler & Felleisen: Contracts for higher-order functions, ICFP ’02]
The Challenge for Lazy Languages

According to
[Deggen, Thiemann, Wehr: *The Interaction of Contracts and Laziness*, PEPM ’12]
- meaning preservation and
- completeness

are contradictory:

\[
\begin{align*}
ep &= \text{assert (pair (prop (== 0)) true) (loop, 42)} \\
main &= \text{print (snd ep)}
\end{align*}
\]
The Challenge for Lazy Languages

According to

[Deggen, Thiemann, Wehr: *The Interaction of Contracts and Laziness, PEPM ’12*]

- meaning preservation and
- completeness

are contradictory:

\[ ep = assert \left( \text{pair} \left( \text{prop} \left( \text{==} \ 0 \right) \right) \text{true} \right) \left( \text{loop}, \ 42 \right) \]

\[ \text{main} = \text{print} \left( \text{snd} \ ep \right) \]

My aim: Meaning preservation but weaker completeness.
High Expressiveness Violates Semantics

Old approach

[Chitil & Huch: *Monadic, prompt lazy assertions in Haskell*, APLAS 2007]
is meaning preserving, but

\[
\begin{align*}
\text{let } x &= \text{assert equal (True,False)} \\
\text{in } (\text{fst } x, \text{snd } x) &\rightsquigarrow \text{exception}
\end{align*}
\]

and

\[
(\text{fst (assert equal (True,False))}, (\text{True,} \\
\text{snd (assert equal (True,False))}) &\rightsquigarrow \text{False})
\]
High Expressiveness Violates Semantics

Old approach

[Chitil & Huch: *Monadic, prompt lazy assertions in Haskell*, APLAS 2007] is meaning preserving, but

\[
\text{let } x = \text{assert equal (True, False)} \quad \Rightarrow (\text{True, error "..."}) \quad \text{or} \\
\text{in (fst x, snd x)} \quad \Rightarrow (\text{error "...", False})
\]

and

\[
\text{(fst (assert equal (True, False)), } \quad \Rightarrow (\text{True,}} \\
\text{snd (assert equal (True, False))) } \quad \Rightarrow \text{False)}
\]

**Hence:** First define semantics, then derive an implementation.
Overview

Part I  Identify contract axioms, derive an implementation.
[Chitil: *A Semantics for Lazy Assertions*, PEPM ’11]

Part II  Consider practical problems for a useful contract library.
[Chitil: *Practical Typed Lazy Contracts*, ICFP 2012]
Lazy Contracts ...

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

```haskell
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!
Axioms of Contracts

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

Definition

Acceptance set $\llbracket c \rrbracket := \{ v \in D \mid \langle c \rangle v = v \} \subseteq D$.

Definition

c is lazy contract, if

1 $\langle c \rangle : D \rightarrow D$ is a continuous function,
2 $c$ is trustworthy, that is, $\langle c \rangle v \in \llbracket c \rrbracket$ for any value $v$,
   (equivalent: $\langle c \rangle$ is idempotent)
3 $\langle c \rangle$ is a partial identity, that is, $\langle c \rangle v \sqsubseteq v$ for any value $v$, and
4 $\llbracket c \rrbracket$ is a lower set.
Definition

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

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

Lemma

$c$ is lazy contract $\iff \langle c \rangle$ is projection and its image is a lower set

cf. [Findler & Blume: *Contracts as pairs of projections*, FLOPS 2006]
Looking for Alternative Axioms

Definition

\[ \downarrow \{ v \} := \{ v' \mid v' \sqsubseteq v \} \]

\[ A_v := \downarrow \{ v \} \cap A \]

Theorem

\[ \llbracket c \rrbracket_v \text{ is an ideal (lower & directed)} \]

\[ \langle c \rangle_v = \bigsqcup \llbracket c \rrbracket_v \]
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 $c$ is a lazy assertion, then $\llbracket c \rrbracket$ is a lazy domain.

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

$$\langle c \rangle v := \bigsqcup \llbracket c \rrbracket_v$$

is a lazy assertion with $\llbracket c \rrbracket = A$. 
Basic Contracts: Minimal & Maximal

Definition

\[
\begin{align*}
[\text{false}] & := \{ \bot \} \\
[\text{true}] & := D
\end{align*}
\]

Derived contract applications

\[
\begin{align*}
\langle \text{false} \rangle v = \biguplus [\text{false}]_v = \biguplus \downarrow \{ v \} \cap \{ \bot \} = \biguplus \{ \bot \} = \bot \\
\langle \text{true} \rangle v = \biguplus [\text{true}]_v = \biguplus \downarrow \{ v \} \cap D = \biguplus \downarrow \{ v \} = v
\end{align*}
\]
**Definition**

\[ [c \& d] := [c] \cap [d] \]

**Lemma** Conjunction is commutative and associative and has \texttt{true} as neutral element.

**Lemma** (Conjunction equals two contracts)

\[ \langle c \& d \rangle v = \langle c \rangle (\langle d \rangle v) \]
Contract Combinators: Disjunction

Not $[c \triangleright d] := [c] \cup [d]$ because $[c \triangleright d]_v = (\downarrow \{v\} \cap [c]) \cup (\downarrow \{v\} \cap [d])$ not directed.

**Definition**

$[c \triangleright d] := \bigcap \{ Y \mid [c] \cup [d] \subseteq Y, Y \text{ lazy domain} \}$

Attention!

$D = \{ \bot, (\bot, \bot), (\text{True}, \bot), (\text{False}, \bot), \ldots, (\text{False}, \text{False}) \}$

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

$[\text{fstTrue} \triangleright \text{sndTrue}] = D$

$[ (\text{fstTrue} \& \text{sndTrue}) \triangleright (\text{fstFalse} \& \text{sndFalse}) ] = D$

**Lemma** Disjunction is commutative and associative and has false as neutral element.
Bounded Distributive Lattice of Contracts

**Lemma (Absorption laws)**

\[ c \land (c \triangleright d) = c \]
\[ c \triangleright (c \land d) = c \]

**Lemma (Distributive laws)**

\[ c \triangleright (d \land e) = (c \triangleright d) \land (c \triangleright e) \]
\[ c \land (d \triangleright e) = (c \land d) \triangleright (c \land e) \]

**Theorem** Lazy contracts form a bounded distributive lattice with meet \( \land \), join \( \triangleright \), least element \textit{false} and greatest element \textit{true}. The ordering is the subset-relationship on acceptance sets.

**Corollary (Idempotency laws)**

\[ c \land c = c \]
\[ c \triangleright c = c \]
Let $[c] := \{\bot, (\bot, \bot)\}$

c & \neg c = \text{false} \implies [c] \cap [\neg c] = \{\bot\}.

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

So $[\neg c] = \{\bot\}$.

But then $[c \mid> \neg c] = [c]$.

Contradiction to $c \mid> \neg c = \text{true}$. 
Deriving an Implementation: Primitive Data Types

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

Definition (Acceptance set of Boolean property contract)

$$[\text{prop } \phi] := \{\bot\} \cup \{v \mid \phi \ v = \text{True}\}$$

Derive application of contract:

$$\langle \text{prop } \phi \rangle \ v = \bigsqcup \downarrow\{v\} \cap [\phi]$$
$$= \bigsqcup \{\bot, v\} \cap (\{\bot\} \cup \{w \mid \phi \ w = \text{True}\})$$
$$= \bigsqcup \{\bot\} \cup (\text{if } \phi \ v \ \text{then } \{v\} \ \text{else } \{\})$$
$$= \text{if } \phi \ v \ \text{then } v \ \text{else } \bot$$

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

\[
\text{prop } \phi & \text{ prop } \psi := \text{prop } (\lambda x. \phi x \land \psi x) \\
\text{prop } \phi \mid> \text{ prop } \psi := \text{prop } (\lambda x. \phi x \lor \psi x)
\]

Verify they work:

\[
\text{[prop } \phi & \text{ prop } \psi] = \text{[prop } \phi] \cap \text{[prop } \psi] = \{ \bot \} \cup \{ v | \phi v \land \psi v \}
\]

\[
\text{[prop } \phi \mid> \text{ prop } \psi] = \bigcap \{ X | \text{[prop } \phi] \cup \text{[prop } \psi] \subseteq X, X \text{ lazy domain}\}
\]

\[
= \bigcap \{ X | \text{[prop } \phi] \cup \text{[prop } \psi] \subseteq X\}
\]

\[
= \text{[prop } \phi] \cup \text{[prop } \psi] = \{ \bot \} \cup \{ v | \phi v \lor \psi v \}
\]

Negation is possible:

\[
\neg(\text{prop } \phi) := \text{prop } (\lambda x. \neg(\phi x))
\]
Recall:

```haskell
data Formula = Imp Formula Formula \mid And Formula Formula \mid 
               Or Formula Formula \mid Not Formula \mid Atom Char
```

```haskell
clausalNF :: Formula \rightarrow \[[Formula]\]
```

```haskell
cClausalNF = assert (conjNF & right \rightarrow list (list lit)) clausalNF
```

So define

```haskell
lit :: Contract Formula
lit = pAtom true \rightarrow pNot (pAtom true)
```

```haskell
list :: Contract a \rightarrow Contract [a]
list c = pNil \rightarrow pCons c (list c)
```
Definition (Acceptance set for pattern contract)

\[
[pC \; c_1 \ldots c_n] := \{\bot\} \cup \{C \; v_1 \ldots v_n \mid v_1 \in [c_1] \ldots v_n \in [c_n]\}
\]

Lemma (Conjunction of constructor assertions)

\[
(pC \; c_1 \ldots c_n) \land (pC \; d_1 \ldots d_n) = pC \; (c_1 \land d_1) \ldots (c_n \land d_n)
\]

\[
(pC \; c_1 \ldots c_n) \land (pC' \; d_1 \ldots d_n) = false \quad \text{if} \quad C \neq C'
\]

Lemma (Disjunction of constructor assertions)

\[
(pC \; c_1 \ldots c_n) \lor (pC \; d_1 \ldots d_n) = pC \; (c_1 \lor d_1) \ldots (c_n \lor d_n)
\]

Also if \( C \neq C' \), then

\[
[(pC \; c_1 \ldots c_n) \lor (pC' \; d_1 \ldots d_n)] = [pC \; c_1 \ldots c_n] \cup [pC' \; d_1 \ldots d_n]
\]
Representation of constructor contract

\[ pC_1 \bar{c}_1 \mid \ldots \mid pC_m \bar{c}_m \]

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

Application of a constructor contract

\[
\langle pC_1 \bar{c}_1 \mid \ldots \mid pC_m \bar{c}_m \rangle (C \overline{v}) = \begin{cases} 
C (\langle \bar{c}_j \rangle \overline{v}) & \text{if } C = C_j \\
\bot & \text{otherwise}
\end{cases}
\]

\[
\langle pC_1 \bar{c}_1 \mid \ldots \mid pC_m \bar{c}_m \rangle \bot = \bot
\]
Algebraic Data Types

Conjunction

\[
(pC_{i_1} \overline{c}_{i_1} \mid \ldots \mid pC_{i_m} \overline{c}_{i_m}) \land (pC_{j_1} \overline{d}_{j_1} \mid \ldots \mid pC_{j_l} \overline{d}_{j_l}) = pC_{k_1} (\overline{c}_{k_1} \land \overline{d}_{k_1}) \mid \ldots \mid pC_{k_o} (\overline{c}_{k_o} \land \overline{d}_{k_o})
\]

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

Disjunction

\[
(pC_{i_1} \overline{c}_{i_1} \mid \ldots \mid pC_{i_m} \overline{c}_{i_m}) \mid (pC_{j_1} \overline{d}_{j_1} \mid \ldots \mid pC_{j_l} \overline{d}_{j_l}) = pC_{k_1} \overline{z}_{k_1} \mid \ldots \mid pC_{k_o} \overline{z}_{k_o}
\]

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

\[
z_{k_s} = \begin{cases} 
\overline{c}_{k_s} \mid \overline{d}_{k_s} & \text{if } k_s \in \{i_1, \ldots, i_m\} \cap \{j_1, \ldots, j_l\} \\
\overline{c}_{k_s} & \text{if } k_s \in \{i_1, \ldots, i_m\}\backslash\{j_1, \ldots, j_l\} \\
\overline{d}_{k_s} & \text{if } k_s \in \{j_1, \ldots, j_l\}\backslash\{i_1, \ldots, i_m\} 
\end{cases}
\]
What about Function Types?

Function contract \( c \to d \)

Eager definition of function contract application:

\[
\langle c \to d \rangle \delta = \lambda x. \langle d \rangle (\delta(\langle c \rangle x))
\]

But

\[
\llbracket c \to d \rrbracket = \{ \delta \mid \langle d \rangle \circ \delta \circ \langle c \rangle = \delta \}
\]

is not a lower set!

Maybe need to relax axiom for function types?
Have

- pure Haskell: language semantics unchanged; portable library
- lazy contracts: preserve program meaning
  
  eager: `assert (list nat) [4,-4] = error "..."`
  lazy: `assert (list nat) [4,-4] = [4, error "..."]`

- a nice algebra of contracts

Still Want

- function type contracts
- simple parametrically polymorphic types
  
  `(&, |>) :: Contract a -> Contract a -> Contract a`

- simple data-type dependent code
  - easy to write by hand
  - can be derived automatically

- when violated, a contract provides information beyond blaming
The Contract API

type Contract a

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

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

true :: Contract a
false :: Contract a

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

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

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

Cf. [Hinze, Jeuring & Löh: Typed contracts for functional programming, FLOPS 2006]
type Contract a = a -> a

assert c = c

prop p x = if p x then x else error "..."

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

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

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

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

Cf. [Findler & Felleisen: Contracts for higher-order functions, ICFP 2002]
We need disjunction of contracts for lazy algebraic data types

\[ (\mid \triangleright) :: \text{Contract } a \rightarrow \text{Contract } a \rightarrow \text{Contract } a \]

for example for

\[
\text{nats :: Contract } [\text{Int}] \\
\text{nats} = \text{pNil} \mid \triangleright \text{pCons } \text{nat} \text{ nats}
\]
Solution

type Contract a = a -> Maybe a

assert c x = case c x of
    Just y  -> y
    Nothing -> error "..."

(c1 |> c2) x = case c1 x of
    Nothing -> c2 x
    Just y   -> Just y

true x  = Just x
false x = Nothing

...
An Algebra of Contracts

Same laws as non-strict `&&` and `||` (not commutative):

\[
\begin{align*}
c_1 \ \& \ (c_2 \ \& \ c_3) &= (c_1 \ \& \ c_2) \ \& \ c_3 \\
true \ \& \ c &= c \\
c \ \& \ true &= c \\
false \ \& \ c &= false \\
... 
\end{align*}
\]

For function contracts:

\[
\begin{align*}
true \ \Rightarrowarrow \ true &= true \\
c_1 \ \Rightarrowarrow \ false &= c_2 \ \Rightarrowarrow \ false \\
(c_1 \ \Rightarrowarrow \ c_2) \ \& \ (c_3 \ \Rightarrowarrow \ c_4) &= (c_3 \ \& \ c_1) \ \Rightarrowarrow \ (c_2 \ \& \ c_4) \\
(c_1 \ \Rightarrowarrow \ c_2) \ \Rightarrowarrow \ (c_3 \ \Rightarrowarrow \ c_4) &= c_1 \ \Rightarrowarrow \ c_2
\end{align*}
\]
**Lemma (Partial identity)**

\[ \text{assert } c \sqsubseteq \text{id} \]

**Claim (Idempotency)**

\[ \text{assert } c \cdot \text{assert } c = \text{assert } c \]
cClausalNF = assert (conjNF & right -> list (list lit)) clausalNF

Contracts:

conjNF, disj, lit, atom, right :: 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

No general negation, but negated pattern contracts pNotImp, ...
Blaming

Implement like eager contracts: blame server or client.

cConst = assert (true -> false -> true) const

true: never blames anybody
false: always blames the client

Different from [Findler & Blume: Contracts as pairs of projections, FLOPS 2006]
On violation report a *path* of data constructors:

*Main> cClausalNF form [[Atom 'a'],[Atom 'b'],Not

*** Exception: Contract at ContractTest.hs:101:3 violated by

((And _ (Or _ (Not {Not _}))))->_

The client is to blame.

- Starting point for debugging.
- Blaming can be wrong: The contract may be wrong.
Derive data-type-dependent code

Derive a contract pattern on demand

\[
\text{conjNF} = $(p \ 'And) \ \text{conjNF} \ \text{conjNF} \ |> \ \text{disj} \\
\text{disj} = $(p \ 'Or) \ \text{disj} \ \text{disj} \ |> \ \text{lit} \\
\text{lit} = $(p \ 'Not) \ \text{atom} \ |> \ \text{atom} \\
\text{atom} = $(p \ 'Atom) \ \text{true}
\]

or declare

\[
$(\text{deriveContracts} \ 'Formula)
\]

Use Template Haskell; other generic Haskell systems
- introduce a class context (Data a)
- cannot handle functions, e.g. inside data structures
Lazy Contracts
- Need lazy pattern combinators (\texttt{pCons}) and disjunction (\texttt{|>}).
- Pattern assertions similar to algebraic data types; subtypes!
- Laziness restricts expressibility!

Semantics
- Few axioms: continuous, trustworthy, partial identity, lower set.
- Acceptance sets $[c]$ are lazy domains, subdomains.
- Algebra of contracts: bounded distributive lattice.

Practice
- \texttt{type Contract a = a -> Maybe a}
- Portable library: \texttt{hackage.haskell.org/package/Contract}

Future
- Dependent function contracts?
- Contracts to express non-strictness properties?
Example Contracts

A predicate contract:

\[
\text{nat} :: \text{Contract } \text{Int} \\
\text{nat} = \text{prop (>= 0)}
\]

Expressing non-strictness of a function:

\[
\text{cLength} = \text{assert (list false } \rightarrow \rightarrow \text{ nat) length}
\]

\[
\text{cConst} = \text{assert (true } \rightarrow \rightarrow \text{ false } \rightarrow \rightarrow \text{ true) const}
\]

A list is not finite:

\[
\text{infinite} :: \text{Contract } [a] \\
\text{infinite} = \text{pCons true infinite}
\]