Typed Lazy Contracts

Olaf Chitil
Assertions

An assertion in C:

```c
#include <assert.h>

list *get_tail (list *ptr)
{
    assert(ptr!=NULL);
    ...
}
```

- specify & dynamically check properties
- more expressive than static types, less effort than verification
- testing with real values
Systematic use of assertions as contract between a server and a client, separating their responsibilities.

A contract in Eiffel:

```eiffel
connect_to_server (server: SOCKET)
  require
    server /= Void and then server.address /= Void
  do
    ...
  ensure
    connected: server.is_connected
end
```

- If pre-condition fails, then method caller is blamed.
- If post-condition fails, then method itself is blamed.

[Bertrand Meyer: *Object-Oriented Software Construction*, Prentice Hall, 1988]
Define new contracts:

```haskell
nat :: Contract Int
nat = prop (>= 0)
```

Define function variants with contracts:

```haskell
cInc = assert (nat -> nat) (+1)
cDec = assert (nat -> nat) (subtract 1)
```

In functional languages contract assertion is a partial identity.

```haskell
\[
\begin{align*}
cInc \ 42 & \Rightarrow 43 \\
cInc \ (-2) & \Rightarrow \text{exception, blame caller} \\
cDec \ 2 & \Rightarrow 1 \\
cDec \ 0 & \Rightarrow \text{exception, blame function}
\end{align*}
\]

where

```haskell
assert :: Contract a -> a -> a
(>->) :: Contract a -> Contract b -> Contract (a->b)
```
Define

cMap = assert ((nat --> nat) --> list nat --> list nat) map

where

map :: (Int --> Int) --> [Int] --> [Int]

and use it:

... cMap (subtract 1) [2,1,0] ...

- When passing functional argument \(\text{subtract 1}\) to \(\text{cMap}\), impossible to check whether it meets the contract.
- When finding that \(\text{subtract 1}\) violates the contract, have to blame call of \(\text{cMap}\), not definition of \(\text{cMap}\).

[Findler & Felleisen: *Contracts for higher-order functions*, ICFP ’02]
I The challenge of designing contracts for lazy languages.

II Design semantics, derive an implementation.
[Chitil: *A Semantics for Lazy Assertions*, PEPM ’11]

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

The Challenge of Designing Contracts for Lazy Languages
Lazy Evaluation

Define the infinite list of Fibonacci numbers:

\[ \text{fibs :: [Integer]} \]
\[ \text{fibs} = 0 : 1 : \text{zipWith (+)} \text{ fibs (tail fibs)} \]

Evaluate:

\[
\begin{align*}
fibs & \\
\leadsto & 0 : 1 : \text{zipWith (+)} \bullet (\text{tail} \bullet) \\
\leadsto & 0 : 1 : \text{zipWith (+)} \bullet \bullet \\
\leadsto & 0 : 1 : 1 : \text{zipWith (+)} \bullet \bullet \\
\leadsto & 0 : 1 : 1 : 2 : \text{zipWith (+)} \bullet \bullet \\
\vdots & \\
\leadsto & 0 : 1 : 1 : 2 : 3 : 5 : 8 : 13 : 21 : \ldots
\end{align*}
\]
Lazy Contracts have to preserve the meaning ...

... of recursively defined data structures:

nats :: Contract [Integer]
nats = list (prop (>= 0))

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

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

- meaning preservation and
- completeness

are contradictory:

\[
x :: (\text{Int}, \text{Int})
\]

\[
x = \text{assert} \ (\text{pair} \ \text{nat} \ \text{nat}) \ (\text{loop}, 42)
\]

\[
\text{loop} = \text{loop}
\]

\[
\text{main} = \text{print} \ (\text{snd} \ x)
\]
A Problem

According to

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

- meaning preservation and
- completeness

are contradictory:

```
x :: (Int, Int)
x = assert (pair nat nat) (loop, 42)

loop = loop

main = print (snd x)
```

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

\[
\text{let } x = \text{assert equal (True, False)} \quad \Rightarrow \quad \text{exception}
\]

\[
\text{in (fst } x, \text{ snd } x) \quad \Rightarrow \quad (\text{True}, \text{False})
\]

and

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

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

is meaning preserving, but

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

and

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

So meaning preservation alone is insufficient.
Part II

Design Semantics, Derive an Implementation
All domains $D$ are directed complete partial order with $\perp_D$ which represents undefinedness and exception for contract failure.

To preserve meaning of recursive definitions of data structures

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

any approximation of an acceptable data structure has to be accepted!

- $\text{assert nats } (0:1: \perp) \rightsquigarrow 0:1: \perp$
- $\text{assert nats } (0:1:1: \perp) \rightsquigarrow 0:1:1: \perp$
- $\text{assert nats } (0:1:1:2: \perp) \rightsquigarrow 0:1:1:2: \perp$
Axioms of Contracts

Let $D$ be a first-order domain, a directed complete partial order with $\bot$. Write $\langle c \rangle : D \to D$ for semantics of assert $c$.

**Definition**

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

**Definition**

$c$ is lazy contract, if

1. $\langle c \rangle : D \to D$ 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

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

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

Theorem

\([c]_v \) is an ideal (lower & directed)

\(\langle c \rangle v = \bigcup [c]_v \)
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 $c$ is a lazy assertion, then $[c]$ is a lazy domain.

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

$$\langle c \rangle v := \bigsqcup [c]_v$$

is a lazy assertion with $[c] = A$. 
Basic Contracts: Minimal & Maximal

Definition

\[
[\text{false}] := \{\bot\} \\
[\text{true}] := \mathcal{D}
\]

Derived contract applications

\[
\langle \text{false} \rangle \nu = \bigcup [\text{false}] \nu = \bigcup \downarrow \{\nu\} \cap \{\bot\} = \bigcup \{\bot\} = \bot
\]

\[
\langle \text{true} \rangle \nu = \bigcup [\text{true}] \nu = \bigcup \downarrow \{\nu\} \cap \mathcal{D} = \bigcup \downarrow \{\nu\} = \nu
\]
Contract Combinators: Conjunction

Definition

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

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

Lemma (Conjunction equals two contracts)

\[ \langle c \& d \rangle \nu = \langle c \rangle (\langle d \rangle \nu) \]
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.
Lemma (Absorption laws)

\[ c \& (c \rightarrow d) = c \]
\[ c \rightarrow (c \& d) = c \]

Lemma (Distributive laws)

\[ c \rightarrow (d \& e) = (c \rightarrow d) \& (c \rightarrow e) \]
\[ c \& (d \rightarrow e) = (c \& d) \rightarrow (c \& e) \]

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

Corollary (Idempotency laws)

\[ c \& c = c \]
\[ c \rightarrow c = c \]
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 \( c \mid> \neg c = \text{true} \).
Flat domain, i.e., \( v \sqsubseteq w \) implies \( v = \bot \).

**Definition (Acceptance set of Boolean property contract)**

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

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:

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

Verify they work:

\[ \left[ \text{prop } \phi \land \text{prop } \psi \right] = \left[ \text{prop } \phi \right] \cap \left[ \text{prop } \psi \right] = \{ \bot \} \cup \{ v \mid \phi(v) \land \psi(v) \} \]

\[ \left[ \text{prop } \phi \mid \text{prop } \psi \right] = \bigcap \{ X \mid \left[ \text{prop } \phi \right] \cup \left[ \text{prop } \psi \right] \subseteq X, X \text{ lazy domain} \} \]
\[ = \bigcap \{ X \mid \left[ \text{prop } \phi \right] \cup \left[ \text{prop } \psi \right] \subseteq X \} \]
\[ = \left[ \text{prop } \phi \right] \cup \left[ \text{prop } \psi \right] = \{ \bot \} \cup \{ v \mid \phi(v) \lor \psi(v) \} \]

Negation is possible:

\[ \neg (\text{prop } \phi) := \text{prop } (\lambda x. \neg(\phi(x))) \]
Pattern Contracts for Algebraic Data Types

Example:

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

Example:

```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

lit :: Contract Formula 
lit = pAtom true |> pNot (pAtom true)

...
Deriving an Implementation: Algebraic Data Types

**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)**

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

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

**Lemma (Disjunction of constructor assertions)**

\[
(pC \ c_1 \ldots c_n) \mid> (pC \ d_1 \ldots d_n) = pC \ (c_1 \mid> d_1) \ldots (c_n \mid> d_n)
\]

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

\[
[(pC \ c_1 \ldots c_n) \mid> (pC' \ d_1 \ldots d_n)] = [(pC \ c_1 \ldots c_n)] \cup [(pC' \ d_1 \ldots d_n)]
\]
Contract Representation and Application

Representation of constructor contract

\[ pC_1 \bar{c}_1 \mid> pC_2 \bar{c}_2 \mid> \ldots \mid> pC_m \bar{c}_m \]

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

Application of a constructor contract

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

\[
\langle pC_1 \bar{c}_1 \mid> \ldots \mid> pC_m \bar{c}_m \rangle \bot = \bot
\]
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 \( \{k_1, \ldots, k_o\} = \{i_1, \ldots, i_m\} \cup \{j_1, \ldots, j_l\} \)

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

Function contract \( c \rightarrow\rightarrow d \)

Just use definition from eager functional languages!

\[
\langle c \rightarrow\rightarrow d \rangle \delta = \langle d \rangle \circ \delta \circ \langle c \rangle
\]

No new primitive contract.
Hence meaning preserving and semantically just a normal function.

Note

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

is not a lower set, but \( \langle c \rightarrow\rightarrow d \rangle \) is a projection.
Part III

Consider Practical Problems for a Useful Contract Library
From Theory to Practice

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

- 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

ture :: 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]
A list is not finite:

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

We need disjunction of contracts for lazy algebraic data types

\[
(|>) :: \text{Contract a -> Contract a -> Contract a}
\]

for example for

\[
\text{nats :: Contract [Int]} \\
\text{nats = pNil |> pCons nat nats}
\]
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):

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

For function contracts:

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

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

Claim (Idempotency)

\[ \text{assert } c \land \text{assert } c = \text{assert } c \]
cClausalNF = assert (conjNF & right \rightarrow list (list lit)) clausalNF

Contracts:

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

conjNF = pAnd conjNF conjNF |\rightarrow disj
disj = pOr disj disj |\rightarrow lit
lit = pNot atom |\rightarrow atom
atom = pAtom true

right = pImp (right & pNotImp) right |\rightarrow
  pAnd (right & pNotAnd) right |\rightarrow
  pOr (right & pNotOr) right |\rightarrow
  pNot right |\rightarrow 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]

Another example expressing non-strictness of a function:

cLength = assert (list false -> nat) length
Add Witness Tracing

On violation report a *path* of data constructors:

```haskell
*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

\[
\begin{align*}
\text{conjNF} & = (p \ 'And) \text{ conjNF conjNF} \rightarrow \text{ disj} \\
\text{disj} & = (p \ 'Or) \text{ disj disj} \rightarrow \text{ lit} \\
\text{lit} & = (p \ 'Not) \text{ atom} \rightarrow \text{ atom} \\
\text{atom} & = (p \ 'Atom) \text{ true}
\end{align*}
\]

or declare

\$(deriveContracts 'Formula)

Use Template Haskell; other generic Haskell systems

- introduce a class context (\textbf{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?