Typed Lazy Contracts

Olaf Chitil

University of Kent

Assertions

An assertion in 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

Contracts

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

A contract in 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]

Contracts in Functional Programming Languages

Define new contracts:

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

Define function variants with contracts:

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

In functional languages contract assertion is a partial identity.

`cInc 42`	$\rightsquigarrow$	43
`cInc (-2)`	$\rightsquigarrow$	exception, blame caller
`cDec 2`	$\rightsquigarrow$	1
`cDec 0`	$\rightsquigarrow$	exception, blame function

where

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

Higher-Order Contracts

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 `(subtract 1)` to `cMap`, impossible to check whether it meets the contract.
- When finding that `(subtract 1)` violates the contract, have to blame call of `cMap`, not definition of `cMap`.

[Findler & Felleisen: *Contracts for higher-order functions*, ICFP '02]

Typed Lazy Contracts

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:

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

Evaluate:

```
    fibs
⇝   0 : 1 : zipWith (+) • (tail • )
⇝   0 : 1 : zipWith (+) • •
⇝   0 : 1 : 1 : zipWith (+) • •
⇝   0 : 1 : 1 : 2 : zipWith (+) • •
⋮         ⋮
⇝   0 : 1 : 1 : 2 : 3 : 5 : 8 : 13 : 21 : ...
```

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

loop = loop

main = print (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

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

$\rightsquigarrow$ exception

and

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

$\rightsquigarrow$

```
(True,
 False)
```

High Expressiveness Violates Semantics

Old approach

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

is meaning preserving, but

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

and

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

So meaning preservation alone is insufficient.

Part II

Design Semantics, Derive an Implementation

Initial Design of Semantics

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

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

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

assert nats (0:1:⊥)	$\leadsto$	0:1:⊥
assert nats (0:1:1:⊥)	$\leadsto$	0:1:1:⊥
assert nats (0:1:1:2:⊥)	$\leadsto$	0:1:1:2:⊥

Axioms of Contracts

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

Definition

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

Definition

c is lazy contract, if

1. $\langle c \rangle : D \to D$ is a continuous function,
2. c is trustworthy, that is, $\langle c \rangle\, v \in [\![c]\!]$ 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. $[\![c]\!]$ is a lower set.

Contracts 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

c is lazy contract $\Leftrightarrow$ $\langle c \rangle$ 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

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

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

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

$$[\![\texttt{false}]\!] := \{\bot\}$$
$$[\![\texttt{true}]\!] := D$$

Derived contract applications

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

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

Contract Combinators: Conjunction

Definition

$$[\![c \mathbin{\&} d]\!] := [\![c]\!] \cap [\![d]\!]$$

Lemma Conjunction is commutative and associative and has `true` as neutral element.

Lemma (Conjunction equals two contracts)

$$\langle c \mathbin{\&} d \rangle \; v = \langle c \rangle \; (\langle d \rangle \; v)$$

Contract Combinators: Disjunction

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

Definition

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

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 is commutative and associative and has `false` as neutral element.

Bounded Distributive Lattice of Contracts

Lemma (Absorption laws)

$$c \mathbin{\&} (c \mathbin{|>} d) = c$$
$$c \mathbin{|>} (c \mathbin{\&} d) = c$$

Lemma (Distributive laws)

$$c \mathbin{|>} (d \mathbin{\&} e) = (c \mathbin{|>} d) \mathbin{\&} (c \mathbin{|>} e)$$
$$c \mathbin{\&} (d \mathbin{|>} e) = (c \mathbin{\&} d) \mathbin{|>} (c \mathbin{\&} e)$$

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

Corollary (Idempotency laws)

$$c \mathbin{\&} c = c$$
$$c \mathbin{|>} c = c$$

No Negation

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

$c \mathbin{\&} \neg c = \texttt{false}$ implies $[\![c]\!] \cap [\![\neg c]\!] = \{\perp\}$.

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

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

But then $[\![c \mathrel{|\!\!>} \neg c]\!] = [\![c]\!]$.

Contradiction to $c \mathrel{|\!\!>} \neg c = \texttt{true}$.

Deriving an Implementation: Primitive Data Types

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

Definition (Acceptance set of Boolean property contract)

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

Derive application of contract:

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

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

Primitive Data Types: Conjunction & Disjunction

Expected definitions:

$$\texttt{prop}\ \phi\ \texttt{\&}\ \texttt{prop}\ \psi := \texttt{prop}\ (\lambda x.\phi\, x \wedge \psi\, x)$$

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

Verify they work:

$$[\![\texttt{prop}\ \phi\ \texttt{\&}\ \texttt{prop}\ \psi]\!] = [\![\texttt{prop}\ \phi]\!] \cap [\![\texttt{prop}\ \psi]\!] = \{\bot\} \cup \{v \mid \phi\, v \wedge \psi\, v\}$$

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

Negation is possible:

$$\neg(\texttt{prop}\ \phi) := \texttt{prop}\ (\lambda x.\neg(\phi\ x))$$

Pattern Contracts for Algebraic Data Types

Example:

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

Example:

```
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)

$$[\![\mathrm{p}C\ c_1 \dots c_n]\!] := \{\bot\} \cup \{C\ v_1 \dots v_n \mid v_1 \in [\![c_1]\!] \dots v_n \in [\![c_n]\!]\}$$

Lemma (Conjunction of constructor assertions)

$$\begin{aligned}(\mathrm{p}C\ c_1 \dots c_n)\ \&\ (\mathrm{p}C\ d_1 \dots d_n) &= \mathrm{p}C\ (c_1 \& d_1) \dots (c_n \& d_n) \\ (\mathrm{p}C\ c_1 \dots c_n)\ \&\ (\mathrm{p}C'\ d_1 \dots d_n) &= \mathtt{false} \qquad \text{if } C \neq C'\end{aligned}$$

Lemma (Disjunction of constructor assertions)

$$(\mathrm{p}C\ c_1 \dots c_n)\ \texttt{|>}\ (\mathrm{p}C\ d_1 \dots d_n) = \mathrm{p}C\ (c_1 \texttt{|>} d_1) \dots (c_n \texttt{|>} d_n)$$

Also if $C \neq C'$, then

$$[\![(\mathrm{p}C\ c_1 \dots c_n)\ \texttt{|>}\ (\mathrm{p}C'\ d_1 \dots d_n)]\!] = [\![\mathrm{p}C\ c_1 \dots c_n]\!] \cup [\![\mathrm{p}C'\ d_1 \dots d_n]\!]$$

Contract Representation and Application

Representation of constructor contract

$$\mathtt{p}C_1\ \overline{c}_1\ \texttt{|>}\ \mathtt{p}C_2\ \overline{c}_2\ \texttt{|>}\ \ldots\ \texttt{|>}\ \mathtt{p}C_m\ \overline{c}_m$$

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

Application of a constructor contract

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

Algebraic Data Types

Conjunction

$$
\begin{aligned}
& (\mathtt{p}C_{i_1}\,\overline{c}_{i_1} \;\mathtt{|>}\; \ldots \;\mathtt{|>}\, \mathtt{p}C_{i_m}\,\overline{c}_{i_m}) \;\&\; (\mathtt{p}C_{j_1}\,\overline{d}_{j_1} \;\mathtt{|>}\; \ldots \;\mathtt{|>}\, \mathtt{p}C_{j_l}\,\overline{d}_{j_l}) \\
= \quad & \mathtt{p}C_{k_1}\,(\overline{c}_{k_1} \& \overline{d}_{k_1}) \;\mathtt{|>}\; \ldots \;\mathtt{|>}\, \mathtt{p}C_{k_o}\,(\overline{c}_{k_o} \& \overline{d}_{k_o})
\end{aligned}
$$

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

Disjunction

$$
\begin{aligned}
& (\mathtt{p}C_{i_1}\,\overline{c}_{i_1} \;\mathtt{|>}\; \ldots \;\mathtt{|>}\, \mathtt{p}C_{i_m}\,\overline{c}_{i_m}) \;\mathtt{|>}\; (\mathtt{p}C_{j_1}\,\overline{d}_{j_1} \;\mathtt{|>}\; \ldots \;\mathtt{|>}\, \mathtt{p}C_{j_l}\,\overline{d}_{j_l}) \\
= \quad & \mathtt{p}C_{k_1}\,\overline{z}_{k_1} \;\mathtt{|>}\; \ldots \;\mathtt{|>}\, \mathtt{p}C_{k_o}\,\overline{z}_{k_o}
\end{aligned}
$$

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

$$
z_{k_s} = \begin{cases}
\overline{c}_{k_s} \,\mathtt{|>}\, \overline{d}_{k_s} & \text{if } k_s \in \{i_1, \ldots, i_m\} \cap \{j_1, \ldots, j_l\} \\
\overline{c}_{k_s} & \text{if } k_s \in \{i_1, \ldots, i_m\} \backslash \{j_1, \ldots, j_l\} \\
\overline{d}_{k_s} & \text{if } k_s \in \{j_1, \ldots, j_l\} \backslash \{i_1, \ldots, i_m\}
\end{cases}
$$

What about Function Types?

Function contract c >-> d

Just use definition from eager functional languages!

$$\langle c \texttt{>->} d \rangle\, \delta = \langle d \rangle \circ \delta \circ \langle c \rangle$$

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

Note

$$[\![c \texttt{>->} d]\!] = \{\delta \mid \langle d \rangle \circ \delta \circ \langle c \rangle = \delta\}$$

is not a lower set, but $\langle c \texttt{>->} d \rangle$ 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

```haskell
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]

A Simple Implementation ...

```
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]

...with a Problem

A list is not finite:

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

We need disjunction of contracts for lazy algebraic data types

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

for example for

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

$$
\begin{aligned}
\texttt{c}_1 \;\texttt{\&}\; (\texttt{c}_2 \;\texttt{\&}\; \texttt{c}_3) &= (\texttt{c}_1 \;\texttt{\&}\; \texttt{c}_2) \;\texttt{\&}\; \texttt{c}_3 \\
\texttt{true} \;\texttt{\&}\; \texttt{c} &= \texttt{c} \\
\texttt{c} \;\texttt{\&}\; \texttt{true} &= \texttt{c} \\
\texttt{false} \;\texttt{\&}\; \texttt{c} &= \texttt{false} \\
&\dots
\end{aligned}
$$

For function contracts:

$$
\begin{aligned}
\texttt{true} \;\texttt{>->}\; \texttt{true} &= \texttt{true} \\
\texttt{c}_1 \;\texttt{>->}\; \texttt{false} &= \texttt{c}_2 \;\texttt{>->}\; \texttt{false} \\
(\texttt{c}_1 \texttt{>->} \texttt{c}_2) \;\texttt{\&}\; (\texttt{c}_3 \texttt{>->} \texttt{c}_4) &= (\texttt{c}_3 \;\texttt{\&}\; \texttt{c}_1) \;\texttt{>->}\; (\texttt{c}_2 \;\texttt{\&}\; \texttt{c}_4) \\
(\texttt{c}_1 \texttt{>->} \texttt{c}_2) \;\texttt{|>}\; (\texttt{c}_3 \texttt{>->} \texttt{c}_4) &= \texttt{c}_1 \;\texttt{>->}\; \texttt{c}_2
\end{aligned}
$$

Contracts are Projections

Lemma (Partial identity)

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

Claim (Idempotency)

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

Contracts for our Earlier Example

```
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]

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:

```
*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

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

or declare

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

Use Template Haskell; other generic Haskell systems

- introduce a class context (`Data a`)
- cannot handle functions, e.g. inside data structures

Last Slide

Lazy Contracts

- Need lazy pattern combinators (`pCons`) and disjunction (`|>`).
- 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

- `type Contract a = a -> Maybe a`
- Portable library: `hackage.haskell.org/package/Contract`

Future

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