Practical Typed Lazy Contracts

Olaf Chitil

University of Kent

10^{th} September
ICFP 2012

Why Contracts?

Specify & dynamically check pre- and post-conditions of functions.

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

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

clausalNF :: Formula -> [[Formula]]
clausalNF (And f1 f2) = cClause f1 : clausalNF f2
clausalNF f           = [cClause f]

cClause = assert (disj & right >-> list lit) clause

clause :: Formula -> [Formula]
clause (Or f1 f2) = f1 : clause f2
clause lit        = [lit]
```

Challenges ⟶ Contributions

- a portable library
- pure Haskell: language semantics unchanged
- simple parametrically polymorphic types

```
list :: Contract a -> Contract [a]
```

- 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
- when violated, contracts provide information beyond blaming
- simple data-type dependent code
    - easy to write by hand
    - can be derived automatically

Example Contracts

A predicate contract:

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

Attaching contracts to functions:

```
cLength = assert (true >-> nat) length

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

Another contract:

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

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

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

For lazy algebraic data types we need disjunction of contracts

```
(|>) :: 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 `||`:

$$
\begin{aligned}
\mathtt{c}_1\ \mathtt{\&}\ (\mathtt{c}_2\ \mathtt{\&}\ \mathtt{c}_3) &= (\mathtt{c}_1\ \mathtt{\&}\ \mathtt{c}_2)\ \mathtt{\&}\ \mathtt{c}_3 \\
\mathtt{true}\ \mathtt{\&}\ \mathtt{c} &= \mathtt{c} \\
\mathtt{c}\ \mathtt{\&}\ \mathtt{true} &= \mathtt{c} \\
\mathtt{false}\ \mathtt{\&}\ \mathtt{c} &= \mathtt{false} \\
&\ldots
\end{aligned}
$$

For function contracts:

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

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 Original 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*

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

Summary

A pure library

- lazy pattern combinators (`pCons`) and disjunction (`|>`)
- `type Contract a = a -> Maybe a`
- contract violation yields location + blame + witness

hackage.haskell.org/package/Contract

Challenge

A lazy dependent function contract:

```
cTake :: Int -> [a] -> [a]
cTake =
  assert (nat >>-> (\n -> lengthAtLeast n >-> listOfLength n))
    take
```