Why Contracts?

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

```haskell
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
  \[ \text{list :: Contract } a \rightarrow \text{Contract } [a] \]
- lazy contracts: preserve program meaning
  \[
  \begin{align*}
  \text{eager: } & \text{assert (list nat) } [4,-4] = \text{error "..." } \\
  \text{lazy: } & \text{assert (list nat) } [4,-4] = [4, \text{error "..."}] 
  \end{align*}
  \]
- 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:

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

Attaching contracts to functions:

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

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

Another contract:

\[
\text{infinite} :: \text{Contract } [a] \\
\text{infinite} = \text{pCons true infinite}
\]
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
A Simple Implementation ...

```haskell
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
```
For lazy algebraic data types we need disjunction of contracts

\((\mid>)\) : Contract a \rightarrow Contract a \rightarrow Contract a

for example for

\texttt{nats :: Contract [Int]}
\texttt{nats = pNil \mid> 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 ||:

\[ c_1 \land (c_2 \land c_3) = (c_1 \land c_2) \land c_3 \]
\[ \text{true} \land c = c \]
\[ c \land \text{true} = c \]
\[ \text{false} \land c = \text{false} \]
\[ \ldots \]

For function contracts:

\[ \text{true} \rightarrow\rightarrow \text{true} = \text{true} \]
\[ c_1 \rightarrow\rightarrow \text{false} = c_2 \rightarrow\rightarrow \text{false} \]
\[ (c_1 \rightarrow\rightarrow c_2) \land (c_3 \rightarrow\rightarrow c_4) = (c_3 \land c_1) \rightarrow\rightarrow (c_2 \land c_4) \]
\[ (c_1 \rightarrow\rightarrow c_2) \mid> (c_3 \rightarrow\rightarrow c_4) = c_1 \rightarrow\rightarrow c_2 \]
Contracts are Projections

Lemma (Partial identity)

assert \( c \sqsubseteq id \)

Claim (Idempotency)

assert \( c \). assert \( c = assert c \)
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, ...
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

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

or declare

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

Use Template Haskell; other generic Haskell systems

- introduce a class context (\text{Data a})
- cannot handle functions, e.g. inside data structures
Summary

A pure library

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

\url{hackage.haskell.org/package/Contract}

Challenge

A lazy dependent function contract:

\begin{verbatim}
cTake :: Int -> [a] -> [a]
cTake = assert (nat >>-> (\n -> lengthAtLeast n >>-> listOfLength n)) take
\end{verbatim}