The Interaction of Contracts and Laziness

Markus Degen, Peter Thiemann, Stefan Wehr

Universität Freiburg, Germany

Shirahama, Japan, 12.04.2010

Design by Contract

- Equip functions with contracts: pre- and postconditions
- Static or dynamic validation
  - static: program verification, theorem proving
  - dynamic: testing, contract monitoring
- Originally proposed for imperative/object-oriented languages
- Extended to higher-order functional languages [Findler, Felleisen 2002]

Contracts for Higher-Order Languages

- Main complication: blame assignment in the presence of higher-order functions
- Non-trivial semantics:
  - projections,
  - pairs of projections,
  - interaction with exceptions
- This work: contracts for lazy functional languages (Haskell)

Contracts for Lazy Functional Languages

Proposals

- Ralf Hinze and Johan Jeuring and Andres Löh. Typed Contracts for Functional Programming. FLOPS 2006.
- Olaf Chitil and Frank Huch. Monadic Prompt Lazy Assertions in Haskell. APLAS 2007.
  (and earlier work by Chitil, McNeill, Runciman)
- Dana N. Xu and Simon Peyton Jones and Koen Claessen. Static Contract Checking for Haskell. POPL 2009.

Contract Language

```
data Ctr :: * -> * where
  Pred :: (a -> Bool) -> Ctr a
  Pair :: Ctr a -> Ctr b -> Ctr (a, b)
  Fun  :: Ctr a -> (a -> Ctr b) -> Ctr (a -> b)

assert :: Ctr a -> a -> a
```

Implementation of Contract Monitoring

[Hinze Jeuring Löh 2006] transcribed and extended from [Findler Felleisen 2002]

```
assert :: Ctr a -> a -> a
assert ctr a = case ctr of
  Pred p     -> if p a then a else error "blame"
  Pair c1 c2 -> (assert c1 (fst a),
                 assert c2 (snd a))
  Fun c1 f2  -> (\ y -> assert (f2 y) (a y)) .
                assert c1
```

Exploration

Exploration

Function `first` selects the first component of a pair

```
first :: (Int, Int) -> Int
first (x, y) = x
```

Given the preconditions `x>y` and `y>=0`, the function `first` returns a strictly positive number.

```
fc :: (Int, Int) -> Int
fc = assert (Fun (Pred (\ (x, y) -> x>y && y>=0))
                 (\ (x, y) -> Pred (\ r -> r>0)))
     first
```

Easy to verify/prove that `first` fulfills this specification.

Example in Strict Haskell

```
fc = assert (Fun (Pred (\ (x, y) -> x>y && y>=0))
                 (\ (x, y) -> Pred (\ r -> r>0)))
     first
```

Imagine a strict variant of Haskell and evaluate

```
fc (-1, 5)
```

- Precondition `x>y` would fail
- Blaming the caller of `fc`

Example in (Lazy) Haskell

Evaluate

```
fc (-1, 5)
```

Expansion of `assert` for the function contract yields

```
let a = (-1, 5)
    a' = assert (Pred (\(x, y) -> x>y && y>=0)) a
    r = assert (Pred (\ r -> r>0)) (first a')
in  r
```

Further expansion of the predicate contract yields

```
let a = (-1, 5)
    a' = if fst a>snd a && snd a>=0
         then a else error "blame caller"
    x = first a'
    r = if x>0 then x else error "blame callee"
in  r
```

Example in (Lazy) Haskell (cont'd)

Result

- contract violation detected
- caller blamed
- **but the semantics is changed**
  - `first (42, let l=l in l)`$\Downarrow$`42`
  - `fc (42, let l=l in l)`$\Uparrow$

Questions

- How severe is the change of the semantics?
- Can it be avoided?
- If so, at what cost?

Lazy Assertions

Lazy Assertions [Chitil Huch 2007] only evaluate a predicate once its arguments are evaluated

- Assertions are evaluated in coroutines
- Implementation involves some cool Haskell hacking

Example in Haskell with Lazy Assertions

```
let (x, y) = (-1, 5)
    -- wait (evaluated x && evaluated y)
    --      (if x>y && y>=0 then ()
    --       else error "blame caller")
    r = x
    -- wait (evaluated r)
    --      (if r>0 then ()
    --       else error "blame callee")
in  r
```

Evaluation of `fc (-1, 5)` yields

- a contract violation

Example in Haskell with Lazy Assertions

```
let (x, y) = (-1, 5)
    -- wait (evaluated x && evaluated y)
    --      (if x>y && y>=0 then ()
    --       else error "blame caller")
    r = x
    -- wait (evaluated r)
    --      (if r>0 then ()
    --       else error "blame callee")
in  r
```

Evaluation of `fc (-1, 5)` yields

- a contract violation
- but shockingly, it now **blames the callee**

Another Example

Consider a slightly different postcondition, which is also implied by the precondition

```
fd :: (Int, Int) -> Int
fd = assert (Fun (Pred (\(x, y) -> x>y && y>=0))
                 (\(x, y) -> Pred (\r -> r>y)))
     first
```

- No difference in strict Haskell
- No difference in the HJL implementation
- What about lazy assertions?

Example Expanded with Lazy Assertions

```
let (x, y) = (-1, 5)
    -- wait (evaluated x && evaluated y)
    --       (if x>y && y>=0 then ()
    -         else error "blame caller")
    r = x
    -- wait (evaluated r && evaluated y)
    --       (if r>y then ()
    --        else error "blame callee")
in  r
```

What happens?

Example Expanded with Lazy Assertions

```
let (x, y) = (-1, 5)
    -- wait (evaluated x && evaluated y)
    --      (if x>y && y>=0 then ()
    -       else error "blame caller")
    r = x
    -- wait (evaluated r && evaluated y)
    --      (if r>y then ()
    --       else error "blame callee")
in  r
```

What happens?

- Neither condition is checked
- `fd` may return *any* integer

Properties of Contract Monitoring

- Meaning preservation / meaning reflection
- Faithfulness / completeness
- Idempotence `(assert c (assert c e))` $\sim$ `assert c e`

Meaning Preservation and Meaning Reflection

MP Adding contracts only adds blame, but does not change the semantics otherwise.

MR Removing contracts does not change successful program runs.

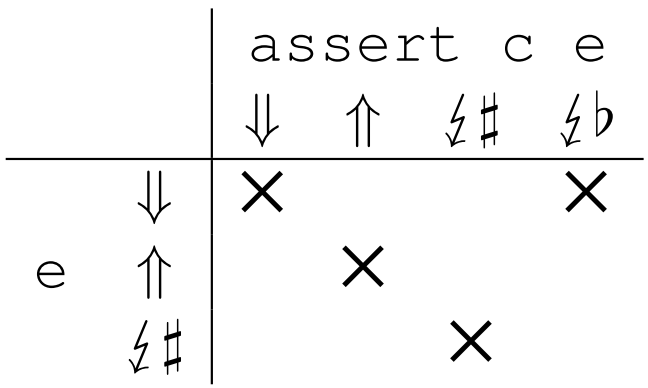
- Both relate evaluation of `e` with `assert c e`

Meaning Preservation and Meaning Reflection

MP Adding contracts only adds blame, but does not change the semantics otherwise.

MR Removing contracts does not change successful program runs.

- Both relate evaluation of `e` with `assert c e`
- Both specify the same relation!



Faithfulness and Completeness

- Express consistency with static verification
- Formalize intuitive expectations

Faithfulness A consumer of a value with an assertion may assume that the assertion and its logical consequences are true.
In particular, the body of a function may assume that the precondition is true and the caller of a function may assume that the postcondition is true for the result.

Completeness Each violation of a contract is detected and signalled by an exception.

Completeness for Predicate Contracts

- Faithfulness and completeness relate the outcome of `p e` with the outcome of `assert (Pred p) e`.
- Both are equivalent!
- Stated in matrix form:

		`assert (Pred p) e`			
		⇓	⇑	↯♯	↯♭
	⇓`False`				×
	⇓`True`	×	×	×	
`p e`	⇑		×		
	↯♯			×	

Results

- Monitoring for strict languages is meaning preserving and faithful (if contracts are assumed to be effect-free)
- HJL monitoring is neither meaning preserving nor faithful
- Lazy assertions are meaning preserving, but not faithful
- Static checking is meaning preserving, but not faithful
- We propose **eager contract monitoring**, which is faithful, but not meaning preserving.
- We conjecture that faithful **and** meaning preserving monitoring for lazy languages is not possible.