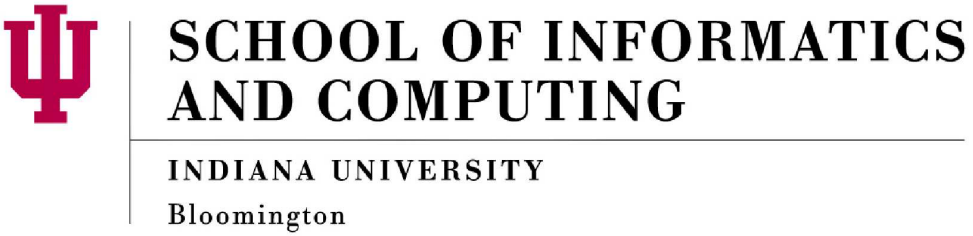
Contract Monitoring as an Effect

Zachary Owens

Indiana University
Bloomington, IN



Higher Order Programming with Effects
September 9, 2012

What are contracts?

- Runtime constraints on values under evaluation
- Express finer-grained intent than most typing systems allow (arguably, contracts are more elegant in most circumstances)
- Framework for ensuring assumptions between module boundaries through ownership and obligation

Contracts in Functional Programming

- Contracts on pure values
- Contracts on pure functions
  - Higher-order function contracts specify a contract on an argument or arguments and the result of the function
  - Dependent contracts are higher-order function contracts in which the contract of the result is dependent on the argument values (Findler and Felleisen 2002)

Racket Contract Example

```
(module A racket
    (provide/contract
      (factorial (→ natural-number/c natural-number/c)))

    (define factorial
      (λ (n)
        (if (zero? n)
            1
            (* n (factorial (sub1 n)))))))

(module B racket
    (require 'A)

    (provide/contract
      (fact5 (and/c natural-number/c (=/c 120))))

    (define fact5 (factorial 5)))
```

Racket Contract Example

```
> (require 'B)
> f5
120
> (require 'A)
> (factorial -1)
factorial: contract violation, expected:
  natural-number/c, given: -1

  contract from: A, blaming: top-level
  contract:
    (→ natural-number/c natural-number/c)
```

Racket Contract Example

```
(module B racket
    (require 'A)

    (provide/contract
      #| changed to (=/c 0) |#
      (fact5 (and/c natural-number/c (=/c 0))))

    (define fact5 (factorial 5)))
```

```
> (require 'B)
self-contract violation, expected:
  (and/c natural-number/c (=/c 0)),
  given 120, which isn't (=/c 0)

  contract from: B, blaming: B
  contract:
    (and/c natural-number/c (=/c 0))
```

Racket Contract Example

```
(module A racket
    (provide/contract
        (factorial (→ natural-number/c natural-number/c)))

    (define factorial
      (λ (n)
        (if (zero? n)
            -1 #| changed to -1 |#
            (* n (factorial (sub1 n)))))))
```

```
> (require 'A)
> (factorial 5)
factorial: self-contract violation, expected:
  natural-number/c, given: -120

  contract from: A, blaming: A
  contract:
    (→ natural-number/c natural-number/c)
```

Coordinating Contracts

- The contract monitor is responsible for handling contract satisfaction
- The monitor is a runtime component
- Abstraction over ownership and obligation
- The monitor is responsible for flagging errors when contracts are violated
- Must keep track of who to blame for a violation (Findler and Felleisen 2002)

Monitor notation

- Monitor is given an expression and a contract for the expression
- Keeps track of blame through 3 labels: positive label, negative label, and contract label (Dimoulas et al. 2011)

$$
\begin{array}{lll}
e & ::= & v \mid x \mid e\ e \mid e + e \mid e - e \mid e \wedge e \mid \\
 & & e \vee e \mid \mathit{zero?}\ e \mid \mathit{if}\ e\ e\ e \mid (e, e) \mid \mathit{fst}\ e \mid \mathit{snd}\ e \mid \\
 & & \mathit{mon}_l^{l,l}\ \kappa\ e \mid \mathit{error}^l \\
 & & \\
v & ::= & 0 \mid 1 \mid -1 \mid \ldots \mid \lambda x.e \mid \mathit{true} \mid \mathit{false} \\
 & & \\
\kappa & ::= & \mathit{flat}(e) \mid \kappa \to \kappa \\
 & & \\
\tau & ::= & o \mid \tau \to \tau \mid \mathit{Contract}\ \tau \\
o & ::= & \mathit{Num} \mid \mathit{Bool} \mid (\tau, \tau)
\end{array}
$$

Monitor Typing Rules

$$\frac{\Gamma \vdash e : o \to \mathit{Bool}}{\Gamma \vdash \mathit{flat}(e) : \mathit{Contract}\ o}$$

$$\frac{\Gamma \vdash \kappa_1 : \mathit{Contract}\ \tau_1 \qquad \Gamma \vdash \kappa_2 : \mathit{Contract}\ \tau_2}{\Gamma \vdash \kappa_1 \to \kappa_2 : \mathit{Contract}\ (\tau_1 \to \tau_2)}$$

$$\frac{\Gamma \vdash \kappa : \mathit{Contract}\ \tau \qquad \Gamma \vdash e : \tau}{\Gamma \vdash \mathit{mon}_j^{k,l}\ (\kappa, e) : \tau}$$

$$\frac{}{\Gamma \vdash \mathit{error}^l : \tau}$$

(Dimoulas et al. 2011)

Desirable Properties of Contracts

Meaning Reflection $mon\ \kappa\ e \Downarrow v \Longrightarrow e \Downarrow v$

Meaning Preservation $e \Downarrow v \Longrightarrow mon\ \kappa\ e \Downarrow v \vee mon\ \kappa\ e \Uparrow error^l$

Faithfulness When contract monitoring is enabled, the predicates that compose the contract are guaranteed to be true.

Idempotence $(mon\ \kappa\ e \Downarrow v \Longrightarrow mon\ \kappa\ (mon\ \kappa\ e) \Downarrow v) \vee$
$(mon\ \kappa\ e \Uparrow error^l \Longrightarrow mon\ \kappa\ (mon\ \kappa\ e) \Uparrow error^l)$

(Degen, Thiemann, and Wehr 2009)

Eager Monitoring

- Functions contracts are delayed
- Value contracts are eager
- Semi-eager monitoring can lead to disaster

Mixing Eager and Delayed Contracts

```
(module contract racket
    (provide C)

    (define pred/c
      (λ (pair)
        (< ((car pair)) ((cdr pair)))))

    (define pair/c
      (cons/c (→ (</c 0)) any/c))

    (define C (and/c pred/c pair/c)))

(module A racket
    (require 'contract)
    (provide/contract (cell C))
    (define cell (cons (λ () 1) (λ () 2))))
```

Mixing Eager and Delayed Contracts

```
> (require 'A)
> cell
'(#<procedure...> . #<procedure...>)
```

Mixing Eager and Delayed Contracts

```
(module contract racket
  (provide C)
  (define pred/c
    (λ (pair)
      (< ((car pair)) ((cdr pair)))))
  (define pair/c (cons/c (→ (</c 0)) any/c))
  (define C (and/c pred/c pair/c)))

(module A racket
  (require 'contract)
  (provide/contract (cell C))
  (define cell (cons (λ () 1) (λ () 2))))

(module B racket
  (require 'A 'contract)

  #| re-export cell with contract C |#
  (provide/contract (cell2 C))
  (define cell2 cell))
```

Mixing Eager and Delayed Contracts

```
> (require 'B)
cell2: self-contract violation, expected:
    (</c 0), given: 1

    contract from: B, blaming: B
    contract:
    (and/c (cons/c (→ (</c 0)) any/c)
           pred/c)
```

Mixing Eager and Delayed Contracts

```
(define pred/c
  (λ (pair)
    (< ((car pair)) ((cdr pair)))))

(define pair/c
  (cons/c (→ (</c 0)) any/c))

(define C (and/c pred/c pair/c))

(define cell (cons (λ () 1) (λ () 2)))
```

1. *mon* pred/c *cell*
2. *mon* pair/c (*mon* pred/c *cell*)
3. *mon* pred/c (*mon* pair/c (*mon* pred/c *cell*)) ⇑ *error*

Idempotent Property

- *mon C e* was applied once and allowed to pass the monitor without a violation
- *mon C* (*mon C e*) was shown to flag a contract violation
- Since *mon C e* and *mon C* (*mon C e*) yielded different results, the contract system is not idempotent

Mixing Eager and Delayed Contracts

```
(module contract racket
  (provide C)
  (define pred/c
    (λ (pair)
      (< ((car pair)) ((cdr pair)))))
  (define pair/c (cons/c (→ (</c 0)) any/c))

  #| Notice the order of the contracts |#
  (define C (and/c pair/c pred/c)))

(module A racket
  (require 'contract)
  (provide/contract (cell C))
  (define cell (cons (λ () 1) (λ () 2))))
```

(Felleisen 2012)

Mixing Eager and Delayed Contracts

```
> (require 'A)
cell: self-contract violation, expected:
    (</c 0), given: 1

    contract from: A, blaming: A
    contract:
    (and/c (cons/c (→ (</c 0)) any/c)
           pred/c)
```

Monitoring as an Effect

The monitoring of contracts is an effect.

1. The monitor has the ability to raise exceptions. (obvious)

2. The monitor changes the order of evaluation of the expression under a contract. Also, expressions that may not have been evaluated could be evaluated because of the intrusive nature of the contract monitor.

Mixing Contracts

- The example in which the idempotence property has broken is an example of a contract calling code that is already under a contract
- Because monitoring a contract for an expression is an effect, the contract is now effectful.
- This violates our original notion of contracts: only pure value contracts or pure function contracts

Appeal of Idempotence

- Racket's immutable data structure contracts are lazy in order to preserve amortized asymptotic behavior of algorithms on these data structures
- Example: binary search tree invariants
- The contract monitor makes a few key optimizations that rely on the idempotence of the contracts on the data structure
  1. Not check redundant contracts
  2. Reduce the overhead of contract checking

(Findler, Guo, and Rogers 2008)

Appeal of Idempotence

- Idempotence holds if conjunction contracts are commutative, namely **and/c** κ_1 κ_2 = **and/c** κ_2 κ_1

- Idempotence is necessary for contracts on general recursive schemes (such as foldr, the list catamorphism)

(Hinze, Jeuring, and Löh 2006)

Monitoring as an Effect

- The effectful nature of the contract monitor is well-known to implementers (Felleisen 2012)
- The effects have never been formalized
- These effects are not obvious to programmers, since the monitor is abstracted
- Contracts on effectful functions are almost impossible to reason about, since the contract monitor's effectfulness is not documented or formalized

Solution to Contract Idempotence

- Must restrict contracts to not be able to write an expression such that adding an additional monitor expression doesn't cause a different result (error or value).

- Solutions:
  - Disallow the idempotence guarantee (Felleisen 2012)
  - Don't allow $\mathit{mon}\ \kappa\ e \Downarrow v \vee \mathit{mon}\ \kappa\ e \Uparrow \mathit{error}^l$ via types. Thus, $\mathit{mon}\ \kappa\ (\mathit{mon}\ \kappa\ e)$ would not be possible. (Degen, Thiemann, and Wehr 2009)
  - Delay the execution of both function *and* predicate contracts (Degen, Thiemann, and Wehr 2009)
  - Use contract monitoring as a logging mechanism rather than a satisfaction mechanism (Disney, Flanagan, and McCarthy 2011)

Monitor Effect at the Type Level

$$e \quad ::= \quad \ldots \mid \mathit{do}\ x \leftarrow e; e \mid \mathit{ret}\ e$$

$$\kappa \quad ::= \quad \ldots \mid \mathit{paircon}(\kappa, \kappa)$$

$$\tau \quad ::= \quad \ldots \mid M\ \tau \mid \mathit{Contract}\ \tau\ \tau$$

Monitor Effect at the Type Level

$$\frac{\Gamma \vdash \kappa : \mathit{Contract}\ \tau_1\ \tau_2 \qquad \Gamma \vdash e : \tau_1}{\Gamma \vdash \mathit{mon}_j^{k,l}\ (\kappa, e) : M\ \tau_2}$$

$$\frac{}{\Gamma \vdash \mathit{error}^l : M\ \tau}$$

$$\frac{\Gamma \vdash e : o \to \mathit{Bool}}{\Gamma \vdash \mathit{flat}(e) : \mathit{Contract}\ o\ o}$$

$$\frac{\Gamma \vdash \kappa_1 : \mathit{Contract}\ \tau_1\ \tau_1' \qquad \Gamma \vdash \kappa_2 : \mathit{Contract}\ \tau_2\ \tau_2'}{\Gamma \vdash \kappa_1 \to \kappa_2 : \mathit{Contract}\ (\tau_1' \to \tau_2)\ (\tau_1 \to M\ \tau_2')}$$

$$\frac{\Gamma \vdash \kappa_1 : \mathit{Contract}\ \tau_1\ \tau_1 \qquad \Gamma \vdash \kappa_2 : \mathit{Contract}\ \tau_2\ \tau_2}{\Gamma \vdash \mathit{paircon}(\kappa_1, \kappa_2) : \mathit{Contract}\ (\tau_1, \tau_2)\ (M\ \tau_1, M\ \tau_2)}$$

Idempotence with Explicit Effectful Types

- *mon* κ (*mon* κ *e*) can no longer be written
- The second call to *mon* must perform the effects of the first call to *mon*

Desirable Properties of Contracts Revisited

Meaning Reflection $\mathit{mon}\ \kappa\ e \Downarrow v \Longrightarrow e \Downarrow v$

Since $\mathit{mon}\ \kappa\ e$ is effectful, this property cannot be guaranteed in general. It would be trivial to prove that for all contracts in e and κ that operate on *pure* functions that this property holds.

Desirable Properties of Contracts Revisited

Meaning Preservation $e \Downarrow v \Longrightarrow \mathit{mon}\ \kappa\ e \Downarrow v \vee \mathit{mon}\ \kappa\ e \Uparrow \mathit{error}^{l}$

This property ignores the fact that the monitor is effectful and can change the order of evaluation.

Desirable Properties of Contracts Revisited

Faithfulness When contract monitoring is enabled, the predicates that compose the contract are guaranteed to be true.

This property is still desirable.

Desirable Properties of Contracts Revisited

Idempotence $(mon\ \kappa\ e \Downarrow v \implies mon\ \kappa\ (mon\ \kappa\ e) \Downarrow v) \vee$
$(mon\ \kappa\ e \Uparrow error^l \implies mon\ \kappa\ (mon\ \kappa\ e) \Uparrow error^l)$

Since $mon\ \kappa\ e$ is effectful in its type this wouldn't type check. However, this may be a useful property. One could write a κ' that is similar to κ, but they make it clear the contract performs effects.

The guarantee of the idempotence property disallows certain contracts that may be nice, such as *montonic*/*c*. (Felleisen 2012)

Contract Monad

- In order to layer additional effects with contract monitoring, a good solution is to use monads and monad transforms for layering additional effects
- The contract monad should still blame the violating module for a contract failure
- The contract monad should still be responsible for halting execution
- In the case of an eager-evaluation language, it should also be responsible for delaying certain expressions such that the monitor is fully delayed
- The advantages is that it is clear from the type that there is an effect

Conclusion

- Contract monitoring is an effect.
- Contract monitoring, in current incarnations, can change the order of evaluation.
- The only way to layer and reason about additional effects is to embrace the effectful nature of the contract monitor.
- Faithfulness and idempotence are the most desirable of contract monitoring properties.

Acknowledgments

- Amr Sabry
- Matthias Felleisen
- Anonymous reviewers on the workshop committee

References I

[1] Markus Degen, Peter Thiemann, and Stefan Wehr. "True Lies: Lazy Contracts for Lazy Languages (Faithfulness is Better than Laziness)". In: *4. Arbeitstagung Programmiersprachen (ATPS'09)*. Lübeck, Germany, 2009.

[2] Christos Dimoulas et al. "Correct blame for contracts: no more scapegoating". In: *SIGPLAN Not.* 46.1 (Jan. 2011), pp. 215–226. ISSN: 0362-1340. DOI: 10.1145/1925844.1926410. URL: http://doi.acm.org/10.1145/1925844.1926410.

[3] Tim Disney, Cormac Flanagan, and Jay McCarthy. "Temporal higher-order contracts". In: *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*. ICFP '11. Tokyo, Japan: ACM, 2011, pp. 176–188. ISBN: 978-1-4503-0865-6. DOI: 10.1145/2034773.2034800. URL: http://doi.acm.org/10.1145/2034773.2034800.

[4] Matthias Felleisen. personal communication. 2012.

[5] Robert Bruce Findler and Matthias Felleisen. "Contracts for higher-order functions". In: *SIGPLAN Not.* 37.9 (Sept. 2002), pp. 48–59. ISSN: 0362-1340. DOI: 10.1145/583852.581484. URL: http://doi.acm.org/10.1145/583852.581484.

References II

[6] Robert Bruce Findler, Shu-Yu Guo, and Anne Rogers. "Implementation and Application of Functional Languages". In: ed. by Olaf Chitil, Zoltán Horváth, and Viktória Zsók. Berlin, Heidelberg: Springer-Verlag, 2008. Chap. Lazy Contract Checking for Immutable Data Structures, pp. 111–128. ISBN: 978-3-540-85372-5. DOI: 10.1007/978-3-540-85373-2_7. URL: http://dx.doi.org/10.1007/978-3-540-85373-2_7.

[7] Ralf Hinze, Johan Jeuring, and Andres Löh. "Typed contracts for functional programming". In: *Proceedings of the 8th international conference on Functional and Logic Programming*. FLOPS'06. Fuji-Susono, Japan: Springer-Verlag, 2006, pp. 208–225. ISBN: 3-540-33438-6, 978-3-540-33438-5. DOI: 10.1007/11737414_15. URL: http://dx.doi.org/10.1007/11737414_15.