Higher-Order Contracts

Cameron Moy

① What are contracts?

② Prehistory: Where did contracts come from? What problem were they trying to solve?

③ History: How are contracts integrated into a programming language? How do you monitor them in a first-order setting?

④ Recent History: What problems arise in higher-order settings?

① A Quick Primer

```
(provide
  (contract-out [fact (→ natural? natural?)]))

(define (fact n)
  (if (zero? n) 1 (* n (fact (sub1 n)))))
```

```
> (fact -1)
fact: contract violation
expected: natural?
given: -1
blaming: REPL
```

← CLIENT BLAME

```
(define (bad-fact n) -1)
> (bad-fact 7)
bad-fact: contract violation
expected: natural?
given: -1
blaming: bad-fact
```

← SERVER BLAME

Myths about Contracts

① "I don't need contracts, I have a type system."

- See fact, need Natural.
- Refinement types? (predicates outside SMT)
- Dependent types? (very hard).

② "Contracts cost too much."

- For fact, contract on export, not internal references.
- An area of active research

③ "Contracts are always checked at run time."

- How they're checked is orthogonal dimension
- Not at all (documentation)
- Statically

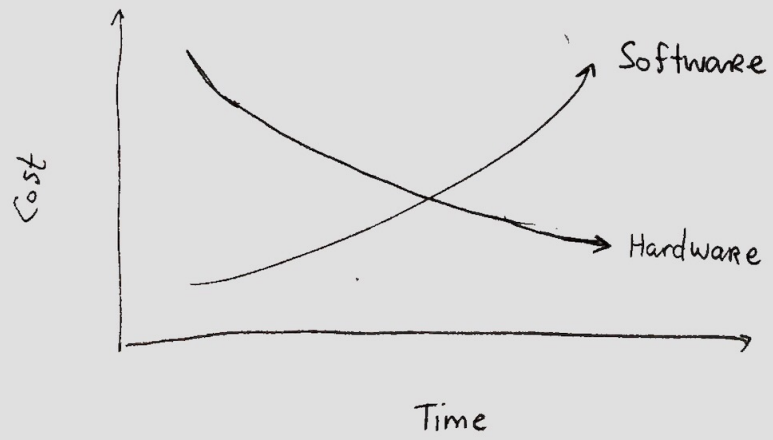
② The Prehistory

Software Crisis of the late 60s.

Large software developments were:

- over-budget
- over-time
- inefficient
- incorrect
- unmaintainable
- ... never even delivered!

Sound familiar?

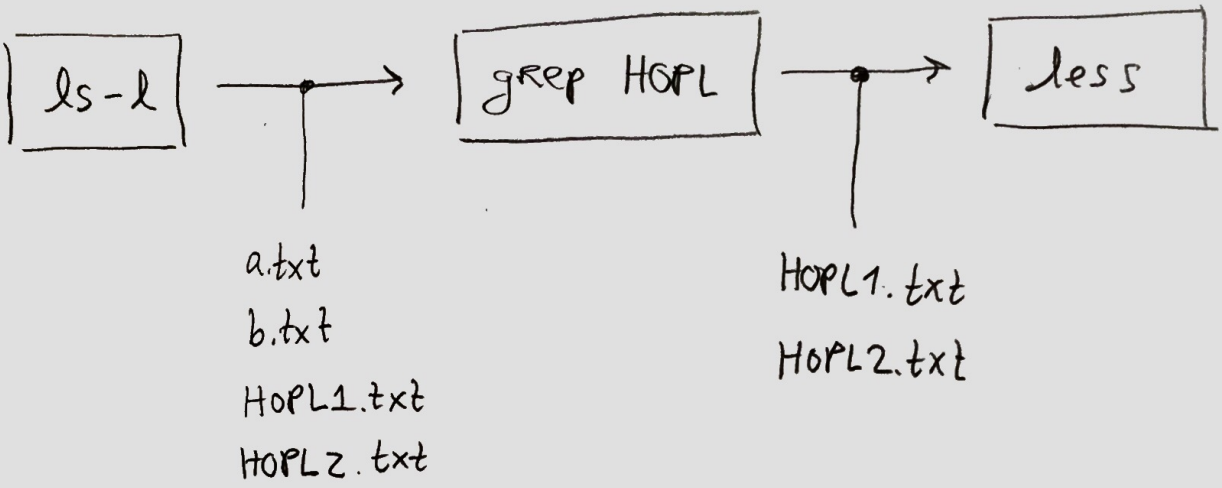


NATO Software Engineering Conference ('68).

McIlroy gives address "Mass Produced Software Components"

- Software isn't industrialized like other fields (e.g. mechanical engineering)
- No "off the shelf" components
- Components may have different characteristics
  - Result precision,
  - algorithm,
  - time-space complexity.
- Need a marketplace of components

> ls -l | grep HOPL | less



- UNIX philosophy of programs doing "one thing"
- Pipes compose these components into a larger unit

- Components are abstract black boxes
- If this is so, how do we know how to use them?
- Informal mechanisms, documentation

"A Technique for Software Module Specification" ('72)

D.L. Parnas

- Components should come with a specification that...
  - describes obligations of clients,
  - describes promises of servers,
  - is reasonably formal;
  - understanable to programmers.
- Specification is systematic (special notation), but inert
- Clients and servers must verify code by hand
- GOAL: Can use component seeing ONLY the spec.

② Contracts in the Language

- Specifications were inert, what if we try to enforce them at run time?
- Known as defensive programming.

```
(define (fact-protected n)
  (if (natural? n)                          ← precondition
      (let ([result (fact n)])
        (if (natural? result)               ← postcondition
            result
            (error "blame server")))
      (error "blame client")))
```

- Disadvantages:
  - Mixes implementation and interface up, Reducing readability
  - Tedious to construct manually, especially good ERROR messages

- What if we synthesize checking code automatically?

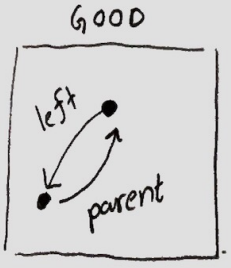
```
fact (n : INT) : INT is
    require   n ≥ 0        ← PRECONDITION
    do
        if n=0 then
            Result := 1.
        else
            Result := n * fact (n-1).
        end
    ensure   Result ≥ 0    ← POSTCONDITION
    end
```

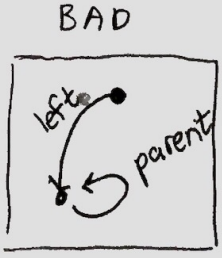
- Eiffel ('86) does this!
- Preconditions and postconditions are assertions (Boolean expressions) on methods
- Language support obviates defensive programming (more advantages like extraction).

- Eiffel is object-oriented and supports more kinds of contracts
- Specifically class invariants

```
class BINARY_TREE [T] feature
  ...
invariant
  left /= Void implies (left.parent = Current);
  right /= Void implies (right.parent = Current)
end.
```

- Invariants are satisfied after construction and preserved by public methods
- Operationally that means adding invariants to precondition and postcondition of every public method (including constructor)





DESIGN BY CONTRACT

- Contracts aren't just "nice to have"
- Central to Meyer's program design methodology
- Component specification essential step (not after-thought)
- Ideally written before implementation
- Cf. "type-driven" development
- Language support makes this style feasible in practice

Is that it??

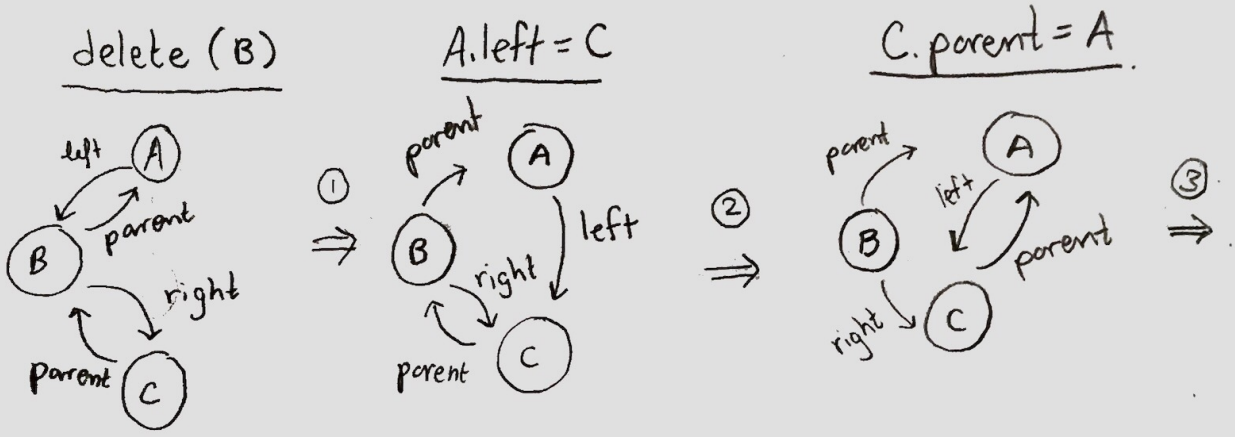
- Pretty simple, just sugar around assertion checking. What's the fuss?
- Contracts as first-order precondition and postcondition checks on procedures — BORING.
- Broad Point: *Language features don't exist in a vacuum.* A new feature interacts with existing ones
  - e.g. type inference

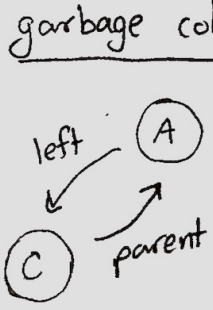
(A) Can't just throw contracts into a realistic language (e.g. OO, functional) and expect things to just work out

(B) Contracts could guarantee more than simple assertions

(A) An OO Complication

- Eiffel recognized some of these issues
- e.g.:





- What happens if there's somehow an exception between steps 1 and 2?
- Class invariant violated at A. ($A.left.parent \neq A$)
- Computing with an inconsistent object state easily yields to undefined behavior
- Eiffel provides rescue clauses to restore invariants in such cases

⑧ Contracts Can say more

- Behavorial Contracts ← What we've seen
- Synchronization
  - e.g. two methods mutually exclusive
- Quality of Service
  - e.g. maximum server response delay
- Security Policies
- Temporal
  - e.g. A is called Before B
- Intensional
  - e.g. A never calls B
- Performance Guarantees

The sky is the limit!

④ The Higher-Order Story

How do real-world languages make contract checking more complex?

Ⓐ Inheritance (OO) } Findler Felleisen

Ⓑ Higher-Order Functions (FP) } Findler Felleisen

Ⓒ Dependent Function Contracts } Dimoulas et al.

Each identifies a limitation or flaw of existing contract systems, then demonstrates how to fix it

Ⓐ Inheritance

```
interface I {
    int m (int a);
    @pre {a > 0}
}
```

```
interface J extends I {
    int m (int a);
    @pre {a > 10}
}
```

```
Class C implements J {
    int m(int a) {
        return sqrt(a-10);
    }
}
```

```
main() {
    I obj = new C();
    obj.m(5);
}
```

- I's precondition OK as 5 > 0
- But not J's!

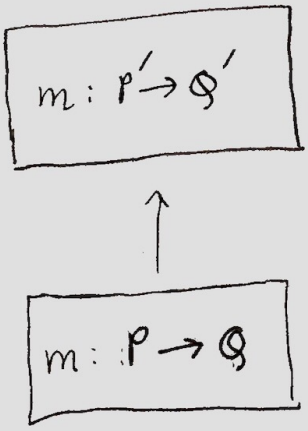
- Liskov Substitution: If J is subtype of I, then objects of type I may be replaced with those of type J.
- We violated this since a C object can't replace an object of type I.

- The problem is the precondition on J is such that implementing classes may not be a <u>behavioral subtype</u> of I
- What can we do?
- What do type systems do?

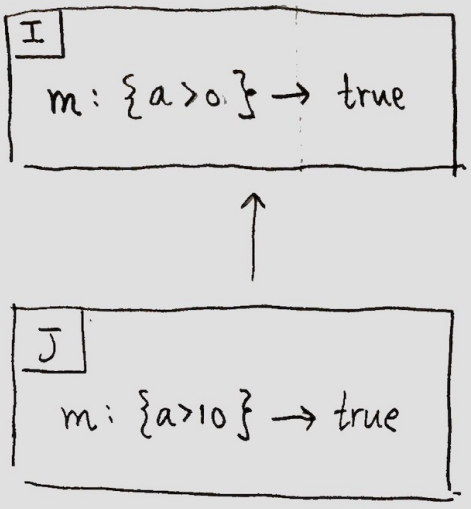
$$\frac{\tau_1' \sqsubseteq \tau_1 \qquad \tau_2 \sqsubseteq \tau_2'}{\tau_1 \to \tau_2 \sqsubseteq \tau_1' \to \tau_2'}$$

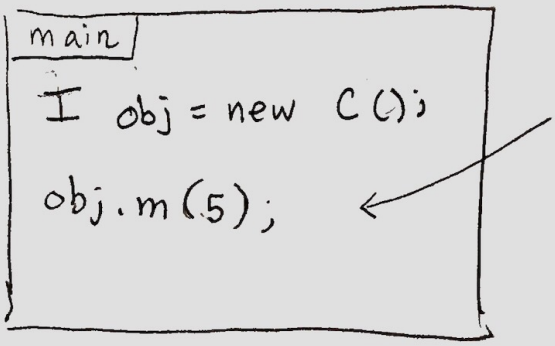
contravariance (↗ $\tau_1' \sqsubseteq \tau_1$) — covariance (↖ $\tau_2 \sqsubseteq \tau_2'$)

- Enforce a similar condition dynamically.



- Thus, $P' \Rightarrow P$ and $Q \Rightarrow Q'$. <u>Should</u> hold.
- How does this check happen in our e.g.?





- $5 > 0$? Yes. (I)
- $5 > 10$? No. (J)
- $a > 0 \Rightarrow a > 10$? No.
  Counterexample, let $a = 5$.

- Consider the following hierarchy:

$$[\!-\!- m : P_1 \to Q_1 -\!\!-] \sqsubseteq \ldots \sqsubseteq [\!-\!- m : P_n \to Q_n -\!\!-]$$

- Suppose $o.m(x)$ with result y.
- Contract system checks:
  1. $P_1(x)$. Otherwise blame client.
  2. $P_n(x) \Rightarrow \ldots \Rightarrow P_1(x)$. Otherwise hierarchy error.
  3. $Q_1(y)$. Otherwise blame server.
  4. $Q_1(y) \Rightarrow \ldots \Rightarrow Q_n(y)$. Otherwise hierarchy error.

- Eiffel "defines the problem away"
  - Precondition P_1, is really $P_1 \vee \bigvee_{k=2}^{n} P_k$
  - Postcondition Q_1, is really $Q_1 \wedge \bigwedge_{k=2}^{n} Q_k$.

(B) Higher-Order Functions

```
(provide
  (contract-out
   [twice (→ (→ natural? natural?) natural?))]))

(define (twice f)
  (f (f 0))).
```


```
(define (twice f)
  (if (something? f)
      (let ([result (f (f 0))])
        (if (natural? result)
            result
            (error "server")))
      (error "client")))
```

- What does something? predicate do?
  How does it check that f is a function natural? → natural?

- A: It can't!
- The predicate can only apply f to a finite number of inputs. But there are infinite naturals.
- Importantly f is opaque black box. Only interaction is function application.
- Solution: Punt!
- Can't check that f satisfies contract now, so we delay the check further in time.
- As long as program never exhibits violation of f's contract on this execution, don't raise ERROR (i.e. need 1st order witness)
- How to delay checks?
- Wrappers (proxies).

```
(define (twice f)
  (f (f 0)))
```

```
(define (twice f)
  (define (f-wrap n)
    (if (natural? n)
        (let ([result (f n)])
          (if (natural? result)
              result
              (error "client")))
        (error "server")))
  (let ([result (f-wrap (f-wrap 0))])
    (if (natural? result)
        result
        (error "server"))))
```

SURPRISE! reversed!

twice : (natural? → natural?) → natural?
① ② ③

---

① (define (twice f)
  (f (f -1)))
(twice sqrt)

} server is at fault

② (define (twice f)
  (f (f 0)))
(twice sub1)

} client is at fault.

③ (define (twice f)
  (sub1 (f (f 0))))
(twice sqrt)

} server is at fault.

Client → Server

(Server → Client) → Server

(Server → Client) → (Client → Server)

⋮

Even-Odd Rule : A first-order contract that appears before an even number of arrows blames the server. Odd blames client

- Among other things, a higher-order contract system will do this kind of wrapping and blame bookkeeping for you.
- Doing this by hand would be infeasible.

- Dependent function contracts also exist.

```
(provide
  (contract-out
    [plus-two  (->d integer?
                    (λ (x)
                      (λ (y)
                        (≤ x y))))]))
```

- contract for an increasing function
- postcondition depends on concrete argument

© Dependent Function Contracts

- Straightforward. Right?
- How to compile this for function g:

```
(define (Q f) (λ (x) (= x (f -1))))
(→d (→ integer? integer?) Q)
```

- Option 1 (Lax):

```
(define (g-protected f)
  (define (f-wrap n)
    ...)
  (let ([result (g f-wrap)])
    (if ((Q f) result)
        result
        (error "server"))))
```

- What about the contract:

```
(→d (→ natural? natural?)
    (λ (f) (λ (x) (= x (f -1)))))
```

- Option 2 (Picky):

```
(define (g-protected f)
  (define (f-wrap x)
    ...)
  (let ([result (g f-wrap)])
    (if ((Q f-wrap) result)
        result
        (error "server"))))
```

in lax this is just f

- Who is blamed for when (f -1) fails in this contract?

```
(→d (→ natural? natural?)
    (λ (f) (λ (x) (= x (f -1)))))
```

- Is that "correct"?
- What does correct even mean?
- How do we know that a given blame strategy is Reasonable?

(→d (→ natural? natural?)
Q) - - - - - - - - - - - - - - - -> imported.

```
module I
(provide Q)
(define (Q f)
  (λ (x)
    (= x (f -1))))
```

- Option 3 (Indy):

```
(define (g-protected f)
  (define (f-wrap n)
    ...)
  (define (f-wrap-indy n)
    (if (natural? n)
        ...
        (error "module I")))
  (let ([result (g f-wrap) I])
    (if ((Q f-wrap-indy) result)
        result
        (error "server"))))
```

different wrappers

- Can we justify this choice of blame assignment over the other two formally?

- Correct Blame: If the contract system blames component C, then that component supplied the bad (contract violating) value.

- Complete Monitoring: The above criterion AND if a value flows from one component to another, its contract must be checked. (Values can't "leak" from one module to another. All communication channels are monitored.)

	Blame Correct	Complete Monitor
Lax	Yes	No
Picky	No	No
Indy	Yes	Yes

- The Complete Monitoring property distinguishes between indy and the others.

Summary

- Contracts specify the assumptions and guarantees of components
- Arose from the software crisis as a way to improve reliability and reusability
- Contracts can be formal and enforced
- Eiffel integrated contracts into the language deeply, yielding many practical benefits
- Contracts in a higher-order setting require care to get right
- Inheritance, higher-order functions, and dependent function contracts all posed interesting challenges
- There's more out there!