What a Great Half-time Show
Congratulations, Pierre-Louis
What a Great Half-time Show
Congratulations, Pierre-Louis

And it has been way too long,
at the opening workshop of PPS,
with a talk on contracts for ho languages
Contracts: Logical Assertions in Component Interfaces

- statically checkable, e.g., types

- behavioral assertions, typically checked at run-time

- temporal assertions, e.g., race conditions, sequencing

- quality of service promises, e.g., # of requests handled
Contracts: Beyond Types

```c
@ensure x >= 0.0
@require sqrt >= 0.0
// compute the square root of x
double sqrt(double x) {
    ...
}
```

impose obligations
make promises

a little bit
Contracts: Beyond Types

```c
@ensure x >= 0.0
@require sqrt >= 0.0
// compute the square root of x
double sqrt(double x) {
    ...
}
```

impose obligations

```c
@ensure x >= 0.0
@require abs(x - sqrt * sqrt)< EPSILON
// compute the square root of x
double sqrt(double x) {
    ...
}
```

make promises

a little bit

a lot more
module mathematics

export: sqrt

module client

import mathematics:
sqrt(-1)
Contracts: Pointing Fingers

module mathematics
export: sqrt

module client
import mathematics:
sqrt(-1)
module mathematics
export: sqrt

module client
import mathematics:
sqrt(-1)

client failed to oblige, blame client
module client
import mathematics:
  sqrt(+1) => -1

module mathematics
export: sqrt

Contracts: Pointing Fingers
module mathematics
export: sqrt

module client
import mathematics:
sqrt(-1)
Contracts: Pointing Fingers

module mathematics
export: sqrt

module client
import mathematics:
  sqrt(+1) => -1

mathematics broke promise, blame mathematics
class ModuleBrowser {

    @require window.topFrame().isTopLevel()
    void mouseHandler(Event e, Posn p) {
        ...
    }
}

state of the world
class Turn {

    private void setPlaced() { ... }
    private boolean isPlacedSet() { ... }

    @require isMergerPosition(p, myBoard())
    @ensure setPlacementCalled()
    void placeHotel(Hotel h, Position p) {
        ...
    }

    @require isPlacedSet()
    Decision retrieveDecision() {
        ...
    }

}
class Turn {

    private void setPlaced() {...} 
    private boolean isPlacedSet() {...} 
    ...

    @require isMergerPosition(p,myBoard())
    @ensure setPlacementCalled()
    void placeHotel(Hotel h, Position p) {
        ...
    }

    @require isPlacedSet()
    Decision retrieveDecision() {
        ...
    }
}

simple temporal
Contracts: Beyond Types

(provide
  (contract-out
;; (grid n m x-delta y-delta x-labels y-labels) creates:
;; -- a graphical grid of size (* n x-delta) x (* m y-delta)
;; -- divvied up into n x m tiles of size x-delta x y-delta
;; -- labeled with x-labels x y-labels
;; -- a function event-handler that deals with mouse clicks for this grid
;; -- (event-handler x y) = (an-x-label, a-y-label)
;; iff (x,y) is a mouse click in the tile labeled (an-x-label, a-y-label)
(grid (->i ((n natural-number/c) (m natural-number/c))
  ((delta-x natural-number/c)
   (delta-y natural-number/c)
   (x-labels list?)
   (y-labels list?)
   #:grid? (grid? #t))
#:pre/name (n x-labels) "correct # of x labels"
(or (unsupplied-arg? x-labels) (= (length x-labels) n))
#:pre/name (m y-labels) "correct # of y labels"
(or (unsupplied-arg? y-labels) (= (length y-labels) m))
(values
  (grid-scene Image?)
  (event-handler
    (delta-x delta-y n m x-labels y-labels)
    (->i ((x natural-number/c) (y natural-number/c))
      #:pre/name (x) "x in proper range"
      (or (unsupplied-arg? delta-x) (< x (* n delta-x)))
      #:pre/name (y) "y in proper range"
      (or (unsupplied-arg? delta-y) (< y (* m delta-y)))
      (values (xl (if (unsupplied-arg? x-labels) any/c (apply or/c x-labels))
        (yl (if (unsupplied-arg? y-labels) any/c (apply or/c y-labels))))))))))

from code base
Contracts: Beyond Types

```
(provide  
(contract-out  
;; Nat Nat [Nat] [Nat] [[Listof X]] [[Listof Y]] -->* Image [Nat Nat -->* X Y]  
;; (grid n m x-delta y-delta x-labels y-labels) creates:  
;; -- a graphical grid of size (* n x-delta) x (* m y-delta)  
;; -- divvied up into n x m tiles of size x-delta x y-delta  
;; -- labeled with x-labels x y-labels  
;; -- a function event-handler that deals with mouse clicks for this grid  
;; (event-handler x y) = (an-x-lablel, a-y-label)  
;; iff (x,y) is a mouse click in the tile labeled (an-x-label, a-y-label)  
(grid -->i ((n natural-number/c) (m natural-number/c))  
((delta-x natural-number/c)  
(delta-y natural-number/c)  
(x-labels list?)  
(y-labels list?)  
#:grid? (grid? #t))  
#:pre/name (n x-labels) "correct # of x labels"  
(or (unsupplied-arg? x-labels) (= (length x-labels) n))  
#:pre/name (m y-labels) "correct # of y labels"  
(or (unsupplied-arg? y-labels) (= (length y-labels) m))  
(values  
(grid-scene image?)  
(event-handler  
(delta-x delta-y n m x-labels y-labels)  
(->i ((x natural-number/c) (y natural-number/c))  
#:pre/name (x) "x in proper range"  
(or (unsupplied-arg? x-labels) (<= x (* n delta-x)))  
#:pre/name (y) "y in proper range"  
(or (unsupplied-arg? delta-y) (<= y (* m delta-y)))  
(values (xl (if (unsupplied-arg? x-labels) any/c (apply or/c x-labels)))  
(yl (if (unsupplied-arg? y-labels) any/c (apply or/c y-labels))))))))))
```
Eiffel is object-oriented.

Objects are legitimate values.

Why are there no contracts on objects?

Or contracts on higher-order functions?
Contracts for Object-Oriented or Functional Languages

Higher-order programming in Eiffel

```java
interface IFunction {
    double apply(double x);
}
```

```java
class Differentiator {
    @ensure ddx.isSlopeOfTangent()
    IFunction ddx(IFunction f) {
        ... 
    }
}
```
Higher-order programming in Eiffel

```eiffel
interface IFunction {
  double apply(double x);
}

class Differentiator {
  @ensure ddx.isSlopeOfTangent()
  IFunction ddx(IFunction f) {
    ...
  }
}
```
Eiffel **flattens** contracts

```java
interface Function {
    double apply(double x);
}

interface IDifferentiated extends Function{
    @ensure isSlopeOfTangent(x)
    double apply(double x);
    boolean isSlopeOfTangent(double x)
}

class Differentiator {
    IDifferentiated d/dx(Function f) {
        ...
    }
}
```
Eiffel’s approach relies on

-- nominal class types
-- duplication of types for contracts
  e.g. int->int must exist once
  per contract attached to D/R
-- closed world of software projects
Contracts for Object-Oriented or Functional Languages

Higher-order contracts in Racket, 1999--2002

```
(define (tangent? f x) ...)
```

```
(provide/contract
 [d/dx
  (->d ((f (-> real? real?)))
    ;; result of d/dx:
    (fprime
     (->d ((x real?))
      ;; result of fprime:
      (y (and/c real? (tangent? f x)))))])
)
```
Racket’s approach imagines

-- an open, growing project

-- with *post-hoc* imposition of contracts on values

-- that is, a *structural* approach to imposing assertions
So what’s the big deal?

(provide/contract
  [d/dx
   (->d ((f (-> real? real?)))
     ;; result of d/dx:
     (fprime
      (->d ((x real?))
        ;; result of fprime:
        (y (and/c real? (tangent? f x))))))]

(define (tangent? f x) ...)
So what’s the big deal?

```
(provide/contract
[d/dx
  (->d ((f (-> real? real?)))
    ;; result of d/dx:
    (fprime
      (->d ((x real?))
        ;; result of fprime:
        (y (and/c real? (tangent? f x)))))])]

(define (tangent? f x) ...)"

You cannot check this property when the function returns.
module mathematics
export: d/dx

module client
import d/dx
f' = d/dx(f)
export: f'

module client3
import f' from client2
f'(z)

module client2
import f' from client
re-export: f'

module client3
import f' from client2
f'(z)
Contracts for Object-Oriented or Functional Languages

module mathematics
export: d/dx

module client
import d/dx
f' = d/dx(f)
export: f'

module client2
import f' from client
re-export: f'

module client3
f' from client2
Contracts for Object-Oriented or Functional Languages

module mathematics
export: d/dx

module client
import d/dx
f' = d/dx(f)
export: f'

module client2
import f' from client
re-export: f'

module client3
f' from client2

blame

Wednesday, September 11, 13
Wrap object with contract checker that carry along source information.
Contracts for Objects and Functions are Complicated

Semantics?
-- what do ho contracts mean?
-- what does blame mean?

Pragmatics!
-- is blame useful/correct?
-- is there “enough” blame?
SEMANTICS
Terms:

\[ e = x \mid \lambda x : t. e \mid e \ e \mid \text{intf}(c,e) \mid \ldots \]

\[ t = b \mid t \to t \]
Terms:

\[ e = x \mid \lambda x: t. e \mid e e \mid \text{intf}(c, e) \mid \ldots \]

\[ t = b \mid t \rightarrow t \]
Terms:

\[ e = x \mid \lambda x : t . e \mid e \cdot e \mid \text{intf}(c, e) \mid \ldots \]

\[ t = b \mid t \rightarrow t \]

Contracts:

\[ c = \text{flat}(\lambda x : b . e) \mid c \rightarrow c \mid c \rightarrow \lambda x : t . c \]
Terms:
\[ e = x \mid \lambda x : t. e \mid e \ y \mid \text{intf}(c, e) \mid \ldots \]
\[ t = b \mid t \to t \]

Contracts:
\[ c = \text{flat}(\lambda x : b. e) \mid c \to c \mid c \to \lambda x : t. c \]

predicates
Terms:

\[ e = x \mid \lambda x: t. e \mid e \cdot e \mid \text{intf}(c, e) \mid \ldots \]
\[ t = b \mid t \rightarrow t \]

Contracts:

\[ c = \text{flat}(\lambda x: b. e) \mid c \rightarrow c \mid c \rightarrow \lambda x: t. c \]

- **predicates**
- **ho contracts**
Terms:
\[ e = x \mid \lambda x : t . e \mid e \ e \mid \text{ intf}(c,e) \mid \ldots \]
\[ t = b \mid t \rightarrow t \]

Contracts:

- **predicates**
  - \[ c = \text{ flat}(\lambda x : b . e) \]

- **ho contracts**
  - \[ c \rightarrow c \]

- **dependent**
  - \[ c \rightarrow \lambda x : t . c \]
C[intf(c,e)]
C[intf(c,e)]

server:
   e

client:
   ...
C[ ---- ]
   ...

Wednesday, September 11, 13
Values flow through contracts, and contracts do their work.
C[ intf(odd?, 5) ]

server:
   e

client:
   ...
   C[ ---- ]

For flat domains, check the value and let it thru if okay.
For flat domains, produce an error if check doesn’t hold.

C[intf(odd?, 8)]

server:

\[
\begin{array}{c}
\text{e} \\
\end{array}
\]

client:

\[
\begin{array}{c}
\text{\ldots} \\
\text{C[ ---- ]} \\
\text{\ldots}
\end{array}
\]
For function domains, ???
C[intf(odd? -> even?, λx:int.2*x)]
\[ C[\text{intf(odd? \rightarrow even?, } \lambda x:\text{int}.2^*x)] \]

For function domains, prune “bad” behavior.

$C[\text{intf}(\text{odd?} \rightarrow \text{even?}, \lambda x:\text{int}.x+1)]$

\[ C[\text{intf}(\text{odd?} \rightarrow \text{even?}, \lambda x: \text{int}.x+1)] \]
C[intf(odd? -> even?, \lambda x:int.x+1)]

Thanks for observably sequential functions and two errors

Cartwright, Curien, Felleisen: IC 1995
Contracts Compose *Pairs of Error Projections in* **ObsSeqfun**

-- the idea works at all contract levels

-- contracts come with a natural order:

\[ c_1 \sqsubseteq c_2 \text{ iff } c_1 = c_1 \circ c_2 \]
\[
(c_1\text{-dom}, c_1\text{-rng}) \sqsubseteq (c_2\text{-dom}, c_2\text{-rng})
\text{ iff } c_2\text{-dom} \sqsubseteq c_1\text{-dom},
\]

and \( c_2\text{-rng} \sqsubseteq c_1\text{-rng} \)

-- contracts are Dana Scott's data types:

\[
\begin{align*}
\text{c1} & \sqsubseteq \text{c2} \\
\text{c2} & \rightarrow \text{c} \quad \text{c} \quad \rightarrow \quad \text{c1} \\
\text{c1} & \sqsubseteq \text{c2} \\
\text{c} & \rightarrow \quad \text{c} \\
\text{c1} & \sqsubseteq \text{c} \\
\text{c} & \rightarrow \quad \text{c2}
\end{align*}
\]
Life is good and practical

-- greatly simplified implementation

-- produced significant performance benefits

-- guided many contract implementation efforts
  • functor-like modules
  • first-class classes
  • mutable objects
  • continuations

Findler & Blume: FLOPS 2004

Findler 2004--present

Strickland & Felleisen, IFL 2008

Strickland et al., TOPLAS 2013

Dimoulas et al., ESOP 2013

Takikawa et al., ESOP 2013
Still the model is imperfect

-- works for flat and higher-order contracts
Still the model is imperfect

-- works for flat and higher-order contracts

-- but it all completely fails for dependent contracts

\[\text{((unit? \rightarrow unit?)} \rightarrow \lambda f:\text{real}\rightarrow\text{real.} (\text{flat}(\lambda x. \ldots(f \ 0)\ldots) \rightarrow \text{unit?}))\]

-- and this kind of code is realistic!
(provide/contract
[d/dx
  (->d ((f (-> real? real?)))
    ;; result of d/dx:
    (fprime
      (->d ((x real?))
        ;; result of fprime:
        (y (and/c real? (tangent? f x)))))]]

(define (tangent? f x) ...)
Probing HO Values is Real

```
(provide/contract
d/dx
  (->d ((f (-> real? real?)))
    ;; result of d/dx:
    (fprime
      (->d ((x real?))
        ;; result of fprime:
        (y (and/c real? (tangent? f x)))))))))

(define (tangent? f x) ...)
```
Probing HO Values is Real

(provide/contract
  [d/dx
    (->d ((f (-> real? real?)))
    ;; result of d/dx:
    (fprime
     (->d ((x real?))
     ;; result of fprime:
     (y (and/c real? (tangent? f x))))))))

(define (tangent? f x) ...)

(provide/contract
  [d/dx
    (->d ((f (-> real? real?)))
    ;; result of d/dx:
    (fprime (-> real? real?))
    #post (test-tangent-randomly fprime f)])

(define (tangent? f x) ...)
Semantics fails to explain pragmatics, especially the proper assignment of blame.
Semantics fails to explain pragmatics, especially the proper assignment of blame.

It hid a lingering bug for over a decade.
PRAGMATICS

From a strictly operational point of view, semantics are unnecessary. To specify a language, we need only give its syntax and pragmatics.

-- James Morris, MIT 1968
PRAGMATICS

~

OPERATIONAL WORKINGS

~

OPERATIONAL SEMANTICS
Pragmatics Means “who’s fault is it”

module mathematics
export: d/dx

module client
import d/dx
f' = d/dx(f)
export: f'

module client3
f' from client2

module client2
import f' from client
re-export: f'

blame
Pragmatics Means “who’s fault is it”

module mathematics
export: d/dx

module client
import d/dx
\( f' = \frac{d}{dx}(f) \)
export: \( f' \)

module client2
import \( f' \) from client
re-export: \( f' \)

module client3
\( f' \) from client2
\[ C[ \text{intf}(\text{odd?} \rightarrow \text{even?}) \rightarrow \lambda f.\text{flat}(f(0) > 0), \lambda f:\text{int} \rightarrow \text{int}.f(1)) ] \]
$$C[\ \text{intf}((\text{odd?} \rightarrow \text{even?}) \rightarrow \lambda f.\text{flat}(f(0) > 0), \lambda f:\text{int}→\text{int}.f(1)) \ ]$$
Who to blame?

Wednesday, September 11, 13
Reductions: No Blame Assignment

\[ \text{intf}(\text{flat}(\lambda x:b.e), v) = \begin{cases} v & \text{if } e(v) \{ v \} \text{ err} \\ \end{cases} \]

\[ \text{intf}(c \to (\lambda x.c'), f) \Rightarrow \lambda x.\text{intf}(c', f(\text{intf}(c, x))) \]
LAX BLAME: Contracts Correct by Definition

\[
\text{intf}[P,C](\text{flat}(\lambda x:b.e), v) \Rightarrow \begin{cases} v & \text{if } e(v) \end{cases} \text{ err}[P]
\]

\[
\text{intf}[P,C](c\rightarrow(\lambda x.c'), f) \Rightarrow \lambda x.\text{intf}[P,C](c', f(\text{intf}[C,P](c, x)))
\]
LAX BLAME: Contracts Correct by Definition

\[
\text{intf}[P,C](\text{flat}(\lambda x:b.e), v) \Rightarrow
\begin{cases}
\text{if } e(v) \{v\} & \text{err}[P]
\end{cases}
\]

\[
\text{intf}[P,C](c \rightarrow (\lambda x.c'), f) \Rightarrow
\lambda x.\text{intf}[P,C](c', f(\text{intf}[C,P](c, x)))
\]

consumers become producers; producers becomes consumers
LAX BLAME: Contracts Correct by Definition

\[ \text{intf}[P,C](\text{flat}(\lambda x:b.e), v) => \]
\[ \text{if } e(v) \{ v \} \text{ err}[P] \]

\[ \text{intf } [P,C](c->(\lambda x.c'), f) => \]
\[ \lambda x.\text{intf}[P,C] c' f(\text{intf}[C,P](c,x)) \]

consumers become producers; producers becomes consumers

the argument x flows into c' without protection
LAX BLAME: Contracts Correct by Definition

\[
\text{intf}[P,C] (\text{flat}(\lambda x:b.e), v) \Rightarrow \\
\quad \text{if } e(v) \{v\} \text{ err}[P]
\]

\[
\text{intf} [P,C] (c\rightarrow(\lambda x.c'), f) \Rightarrow \\
\quad \lambda x.\text{intf}[P,C] c' f (\text{intf}[C,P] (c, x)) 
\]

consumers become producers; producers becomes consumers

the argument \( x \) flows into \( c' \) without protection

Findler & Felleisen: ICFP 2002
PICKY BLAME: Contracts May Violate Contracts

\[
\text{inf}_{P,C} (\text{flat}(\lambda x:b.e), v) \Rightarrow \\
\quad \text{if } e(v) \{ v \} \text{ err}[P]
\]

\[
\text{inf}_{P,C} (c \mapsto (\lambda x.c'), f) \Rightarrow \\
\quad \lambda x. \text{inf}_{P,C} (c' \{ x:=\text{inf}_{P,C} (c,x) \}, \\
\qquad f (\text{inf}_{C,P} (c,x))
\]
Theorem (Greenberg, Pierce, & Weirich, POPL 2010)

The *picky* pragmatics detects all the contract violations that the *lax* pragmatics discovers and then some.
Theorem (Greenberg, Pierce, & Weirich, POPL 2010)

The *picky* pragmatics detects all the contract violations that the *lax* pragmatics discovers and then some.

But *picky* may blame the wrong component.
Theorem (Greenberg, Pierce, & Weirich, POPL 2010)

The *picky* pragmatics detects all the contract violations that the *lax* pragmatics discovers and then some.

But *picky* may blame the *wrong* component.
**Definition** (Dimoulas, Findler, Flanagan, Felleisen, POPL 2011)

M may be blamed for a contract violation iff it controls the flow of a value through a contract and the value violates a part of the contract that belongs to M’s obligations.

**Theorem** (Dimoulas, Findler, Flanagan, Felleisen, POPL 2011)

The *lax* pragmatics never blames the wrong module; *picky* may blame the wrong module.

But *lax* may fail to discover a contract violation.
**Definition** (Dimoulas, Findler, Flanagan, Felleisen, POPL 2011)

M may be blamed for a contract violation iff it controls the flow of a value through a contract and the value violates a part of the contract that belongs to M’s obligations.

**Theorem** (Dimoulas, Findler, Flanagan, Felleisen, POPL 2011)

The *lax* pragmatics never blames the wrong module; *picky* may blame the wrong module.

But *lax* may **fail** to discover a contract violation.
Definition (Dimoulas, Tobin-Hochstadt, Felleisen, ESOP 2012)

A contract system fails to discover a contract violation if a bad value can migrate from one module to another without flowing through a contract boundary.
**Definition** (Dimoulas, Tobin-Hochstadt, Felleisen, ESOP 2012)

A contract system fails to discover a contract violation if a bad value can migrate from one module to another *without flowing through a contract boundary*.

**Theorem** (Dimoulas, Tobin-Hochstadt, Felleisen, ESOP 2012)

The *picky* pragmatics fails to discover contract violations, but *lax* may allow values to slip through w/o checking.
Definition (Dimoulas, Tobin-Hochstadt, Felleisen, ESOP 2012)

A contract system fails to discover a contract violation if a bad value can migrate from one module to another without flowing through a contract boundary.

Theorem (Dimoulas, Tobin-Hochstadt, Felleisen, ESOP 2012)

The picky pragmatics fails to discover contract violations, but lax may allow values to slip through w/o checking.

Neither lax nor picky are “good” pragmatics.
Neither lax nor picky are “good” pragmatics.
INDY BLAME: Contracts *May Blame Contracts*

\[
\text{intf}[P,C,K] (\text{flat}(\lambda x: b. e), v) \Rightarrow \\
\quad \text{if } e(v) \{ v \} \text{ err}[P]
\]

\[
\text{intf}[P,C,K] (c \rightarrow (\lambda x. c'), f) \Rightarrow \\
\quad \lambda x. \text{intf}[P,C,K] (c' \{ x := \text{intf}[P,K,K] (c, x) \}, f(\text{intf}[P,C,K] (c, x))
\]
INDY BLAME: Contracts  

` intf[P,C,K](flat(λx:b.e), v) => if e(v){v} err[P]`

` intf[P,C,K](c->(λx.c'), f) => λx.intf[P,C,K](c'{x:= intf[P,K,K](c,x)}, f(intf[P,C,K](c,x)))`

contracts are *independent* components, and they become consumers
Theorem \textit{(Dimoulas et al.: POPL 2011, ESOP 2012)}

The \textit{indy} pragmatics detects all contract violations and assigns blame to a module that controls the flow of the bad value.
Theorem (Dimoulas et al.: POPL 2011, ESOP 2012)

The *indy* pragmatics detects all contract violations and assigns blame to a module that controls the flow of the bad value.

And there is no “but” other than the proof is horribly complex.
Life is perfect

-- correct blame uncovered bugs in our implementation

-- complete monitoring became our design guide

-- correct blame with complete monitoring

together have become the foundation of our
Typed Racket (gradual typing) research program
CONCLUSIONS
Two Semantic Frameworks for HO Contracts

• SPCF’s semantics of “concrete data structures”
  • it is elegant theory
  • it’s imperfect but easy-to-use

• perfect reduction pragmatics
  • with an ugly subject reduction proof
  • but practical implications
So thanks again for CDS, SPCF, and two errors.