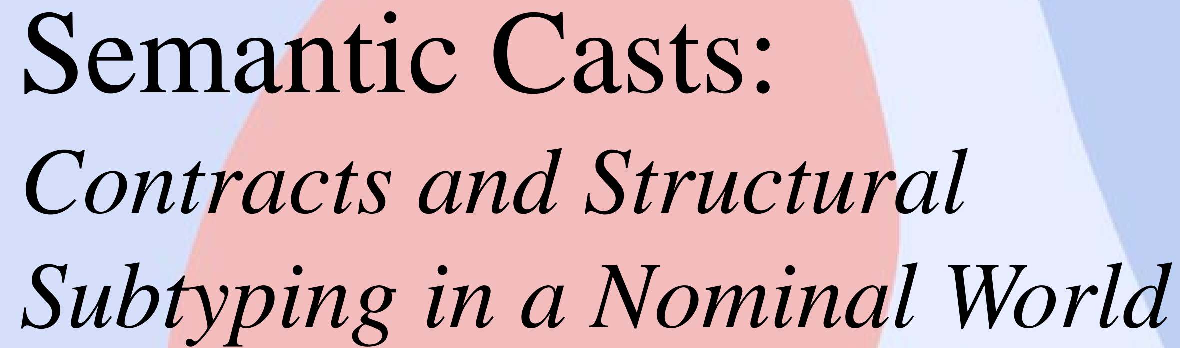


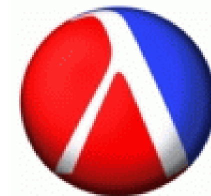
Semantic Casts:

Contracts and Structural Subtyping in a Nominal World

Robby Findler
University of Chicago

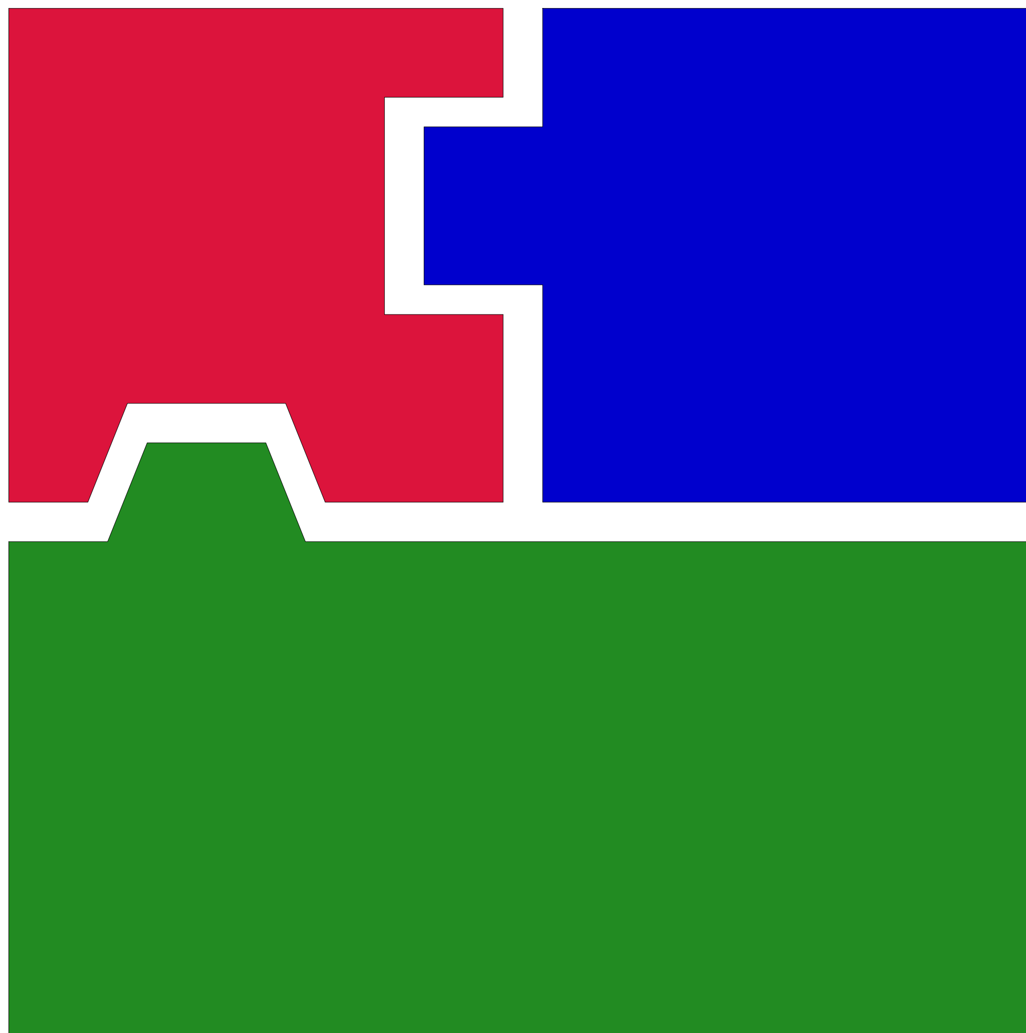
Matthew Flatt
University of Utah

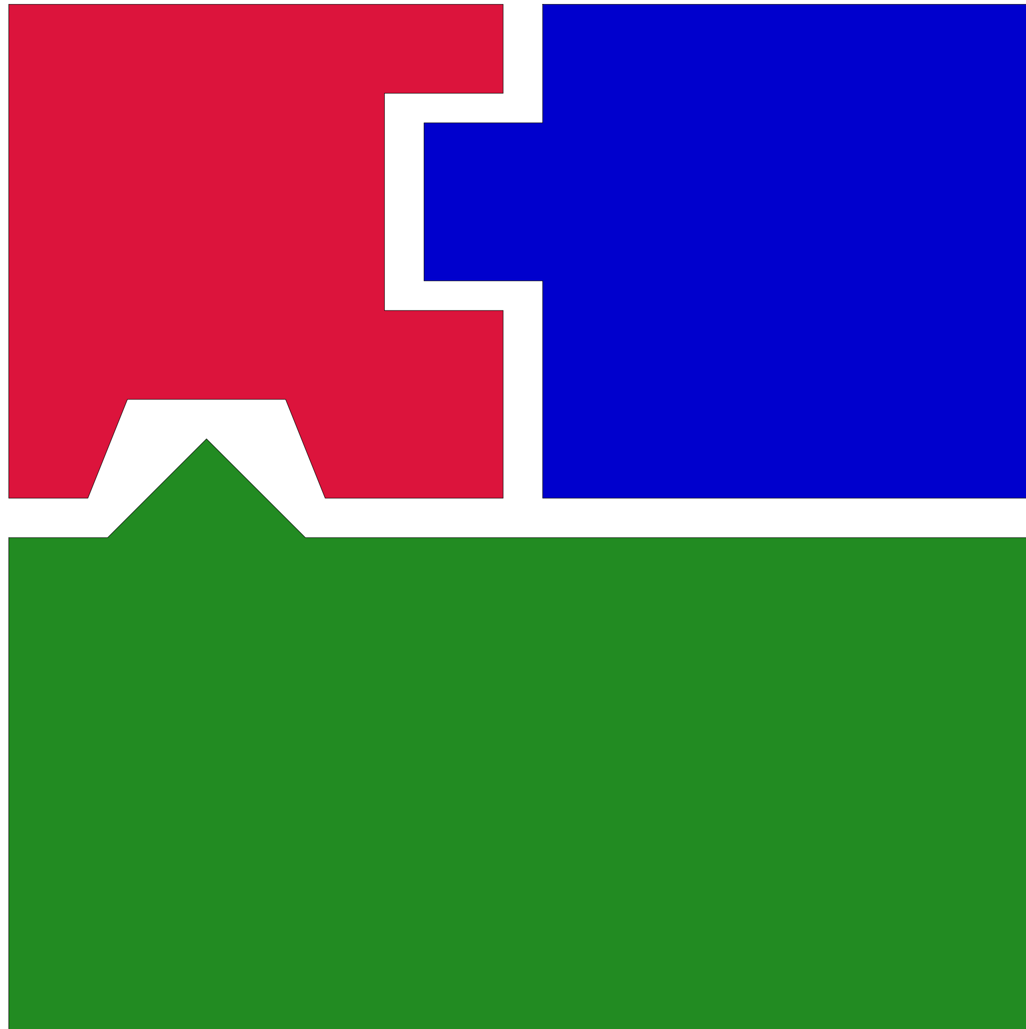
Matthias Felleisen
Northeastern University

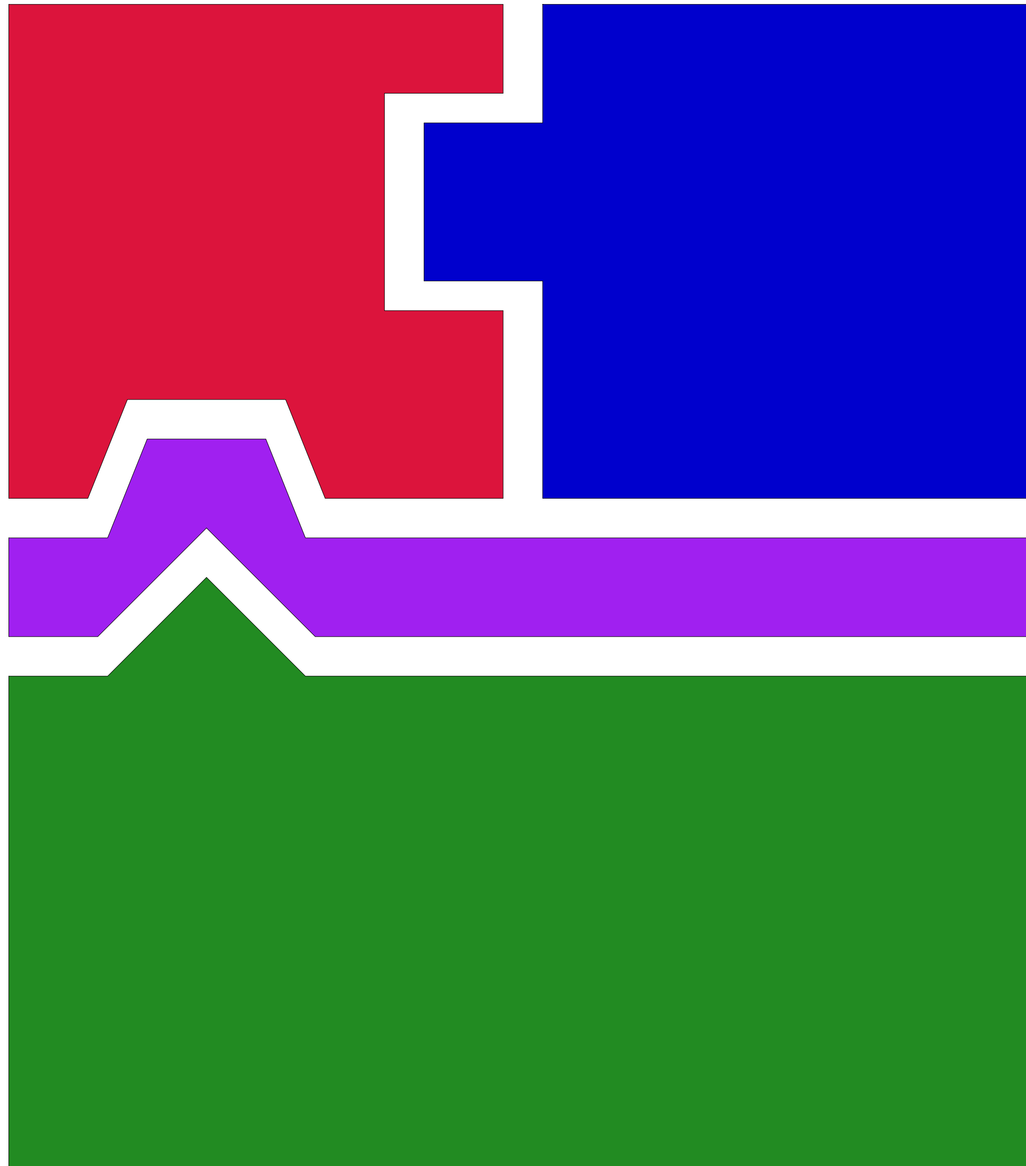


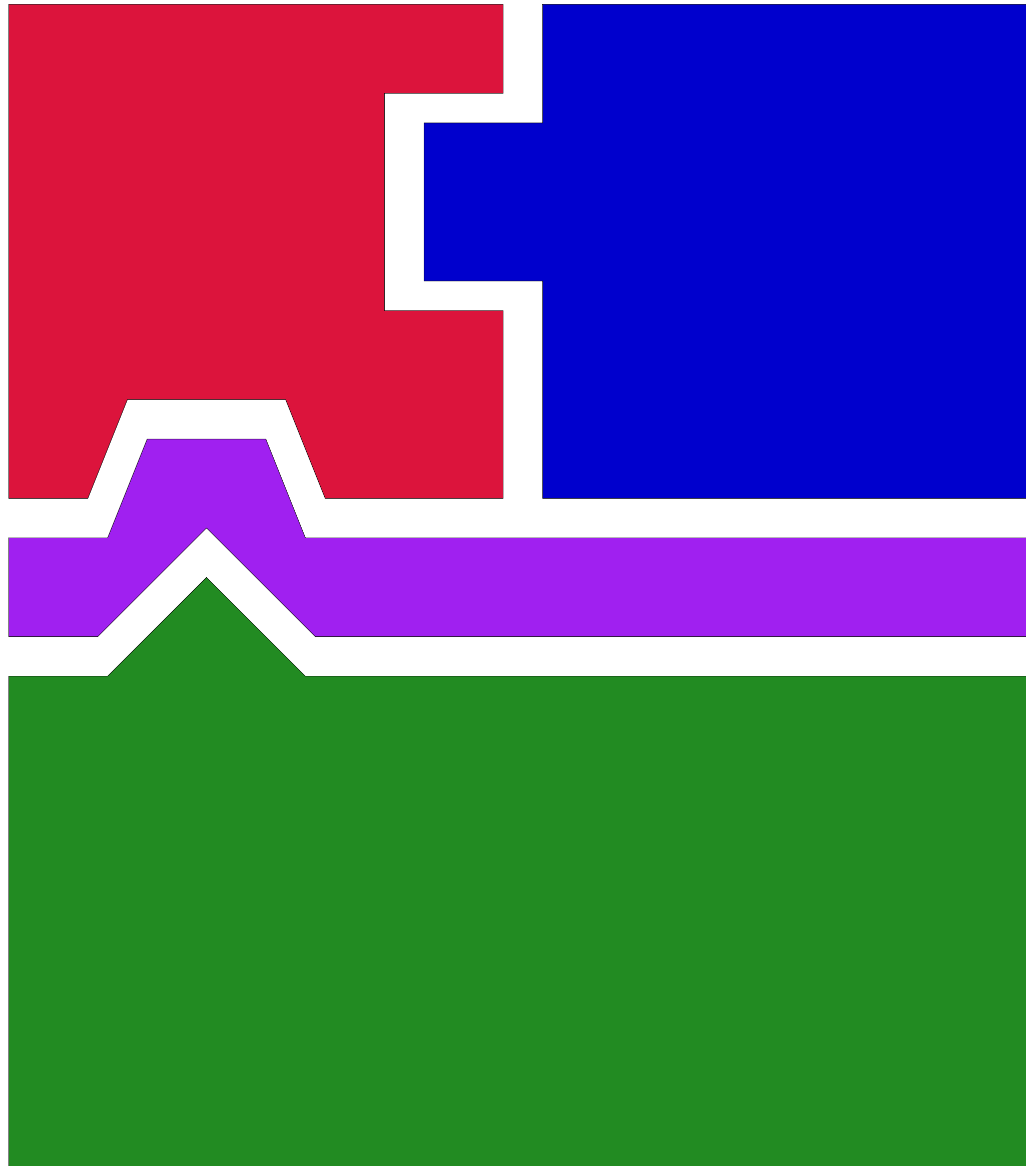
Component marketplace

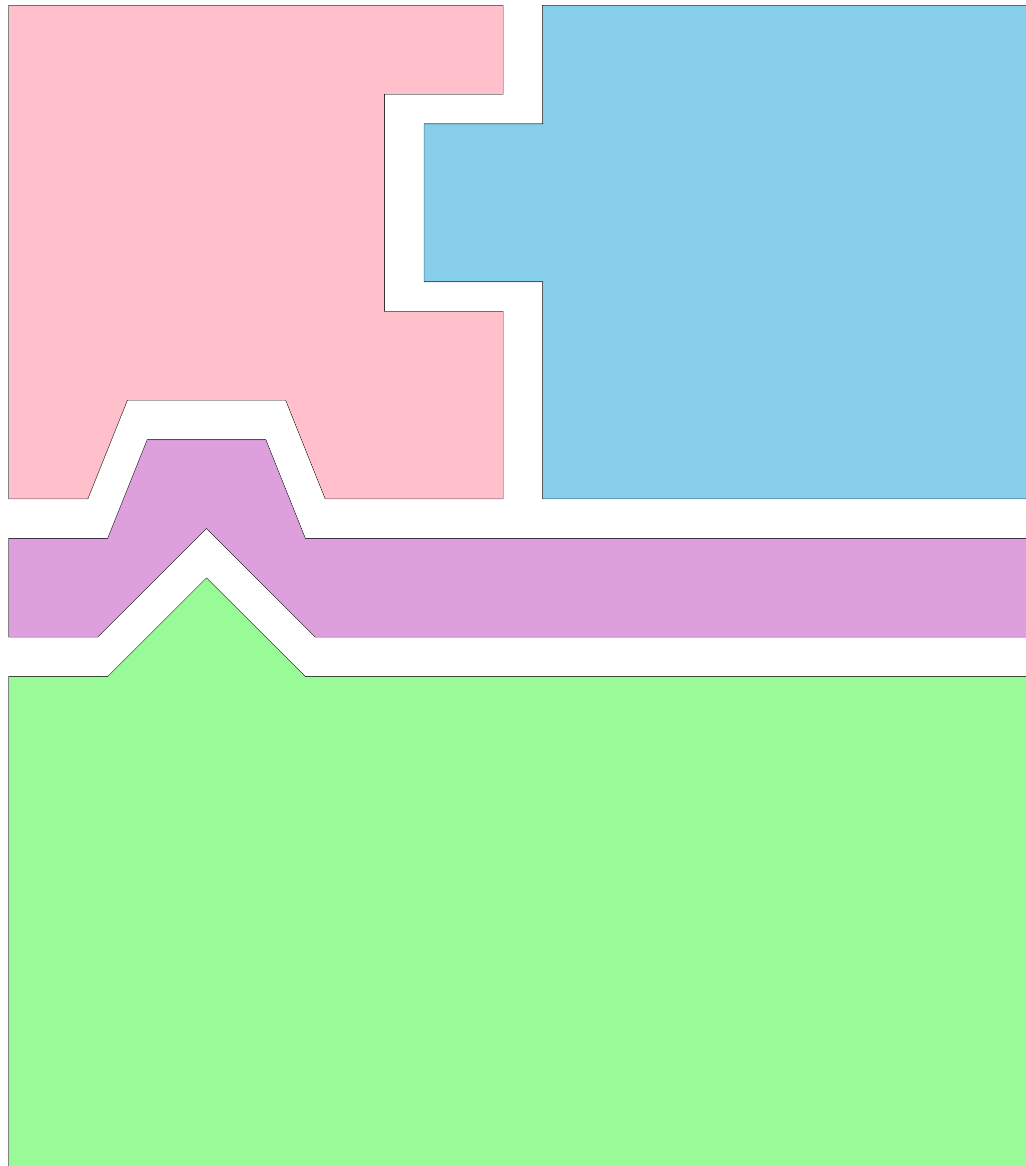
- McIlroy's vision (1969)
- Independent developers produce pieces of programs (components)
- 3rd parties compose the components
- Economic benefits: division of labor and competition
- Software construction: merely plug & play

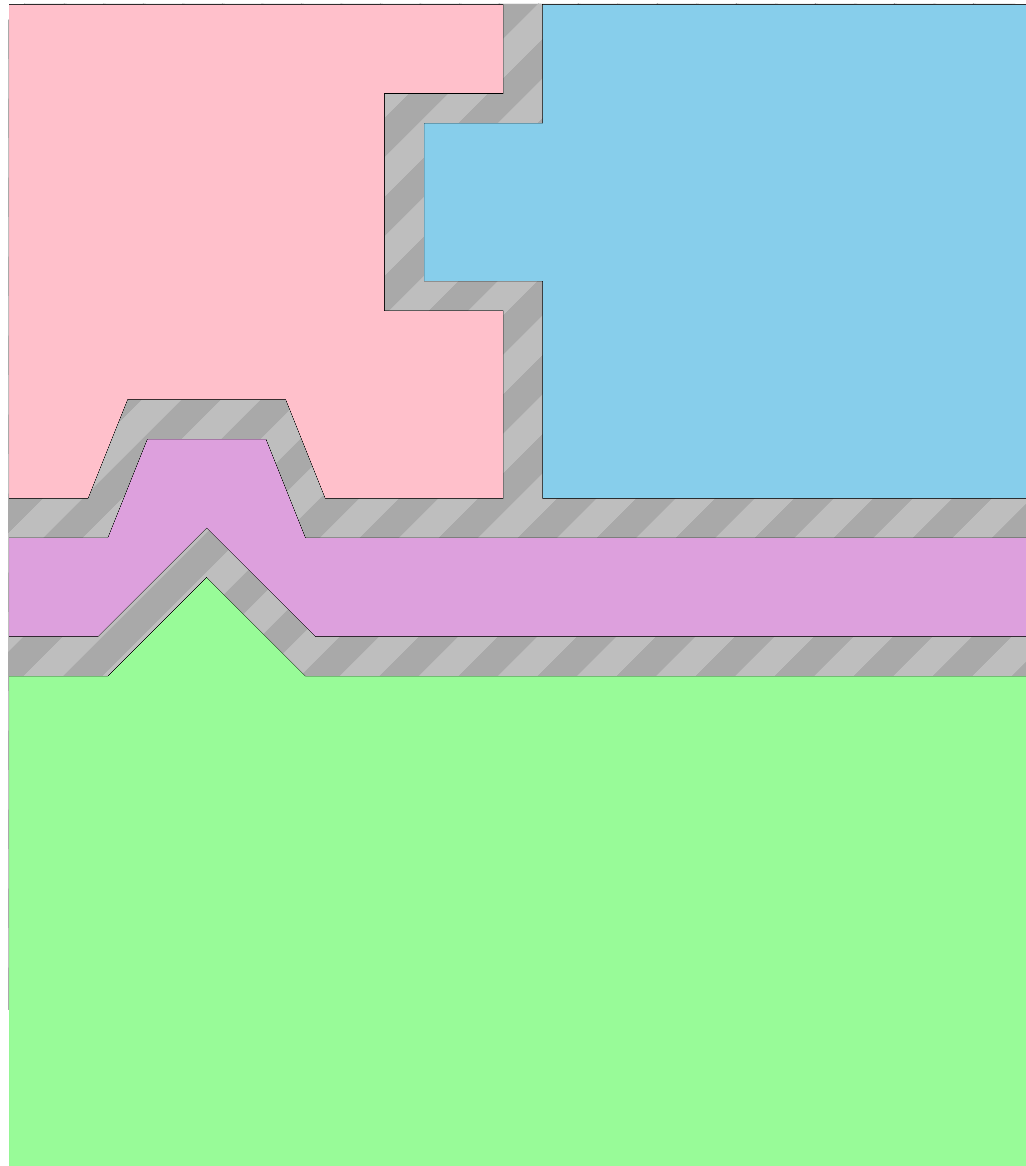


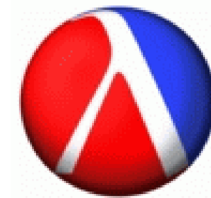






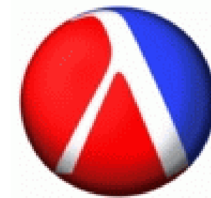






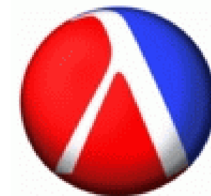
What is a contract?

- Agreement between two components
- Only allows certain patterns of interactions



Why check contracts?

- Find faulty components
- Accountability supports component economy



Contracts [Beugnard et al. 1999]

- Syntactic: types

  int f(int[] x, int k)

- Semantic level 1: behavioral

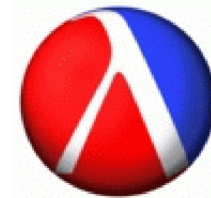
  int f(int[] x, int k) // 0 <= k < x.length()

- Semantic level 2: sequencing, concurrency

  finalize is called for all objects

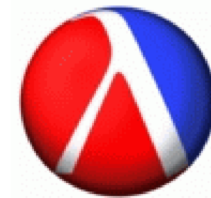
- Quality of service: space, time

  web server handles at least 1000 GET/sec



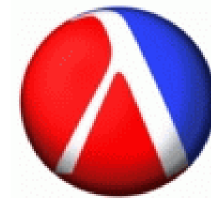
Behavioral contract desiderata

- Simplicity (dynamic enforcement)
- Precise enforcement (no false pos/neg)
- Blame

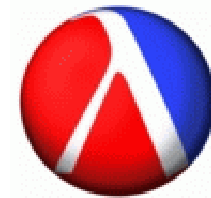


Behavioral contract history

- Parnas: 1972
- Luckham: ANNA for Ada
- Meyer: Eiffel
- ...

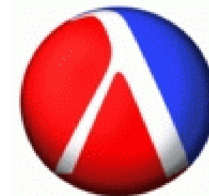


Queues: an example



Queue

```
class Q implements IQueue {
  void enq(int X) {...}


  int deq() {...}


  boolean empty() {...}
}
```



Queue

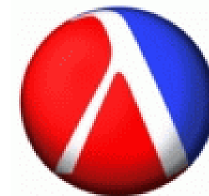
```
class Q implements IQueue {
  void enq(int X) {...}
  // @post !this.empty()

  int deq() {...}
  // @pre !this.empty()

  boolean empty() {...}
}
```

Queue

```
class Q implements IQueue {
  void enq(int X) {...}
  // @post !this.empty()

  int deq() {...}
  // @pre !this.empty()

  boolean empty() {...}
}
```

Good client

```
IQueue q = new Q();
q.enq(1);
q.deq();
```



Queue

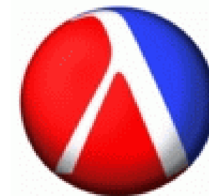
```
class Q implements IQueue {
  void enq(int X) {...}
  // @post !this.empty()

  int deq() {...}
  // @pre !this.empty()

  boolean empty() {...}
}
```

Good client

```
IQueue q = new Q();
q.enq(1);
q.deq();
```

Bad client

```
IQueue q = new Q();
q.deq();
q.enq(1);
```



Queue

```
class Q implements IQueue {
  void enq(int X) {...}
  // @post !this.empty()

  int deq() {...}
  // @pre !this.empty()

  boolean empty() {...}
}
```

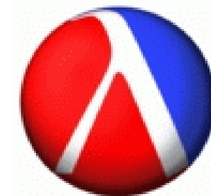
Good client

```
IQueue q = new Q();
q.enq(1);
q.deq();
```

Bad client

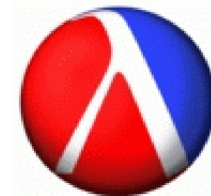
```
IQueue q = new Q();
q.deq();
q.enq(1);
```

Blame q.deq(); in Bad Client



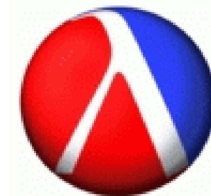
Queue with observer

```
class Q implements IQueue {
  Obs o;

  void enq(int X) {...}
  // @post !this.empty()
  // effect: o.onEnq(this)

  int deq() {...}
  // @pre !this.empty()
  // effect: o.onDeq(this)

  void register(Obs _o) {o = _o;}
  // please: a "good" Observer
}
```



Good observer

```
class GoodO
  implements Obs {
  void init() {...}

  void onEnq(IQueue q)
     {...}
  // @post !q.empty()

  void onDeq(IQueue q)
     {...}
}
```

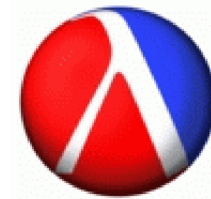


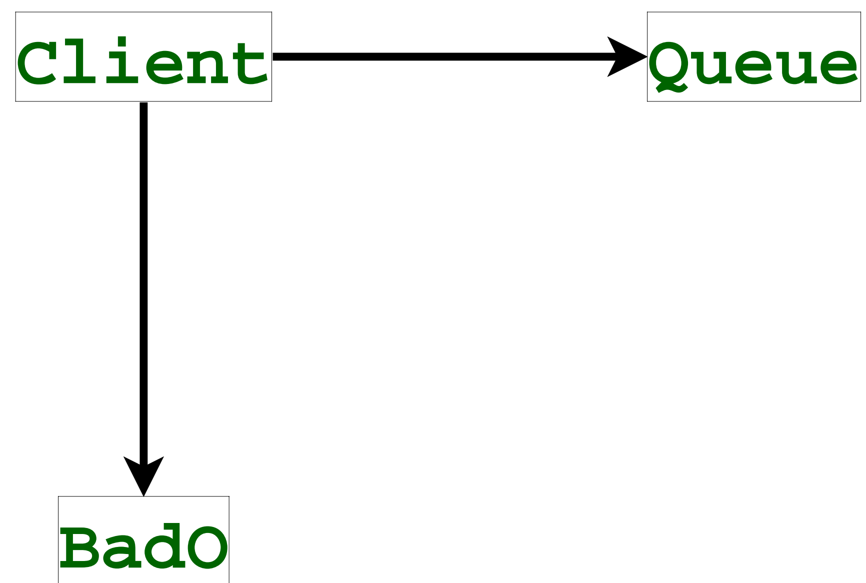
Good observer

```
class GoodO
  implements Obs {
  void init() {...}

  void onEnq(IQueue q)
      {...}
  // @post !q.empty()

  void onDeq(IQueue q)
      {...}
}
```

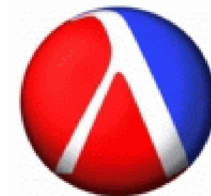
Bad Observer

```
class BadO
  implements Obs {
  void init() {...}

  void onEnq(IQueue q)
    { q.deq() }


  void onDeq(IQueue q)
    {...}
}
```

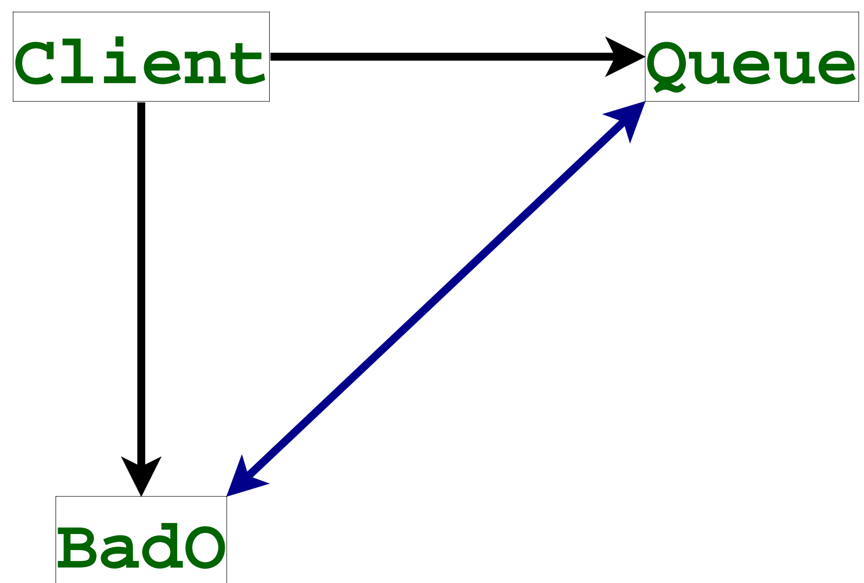


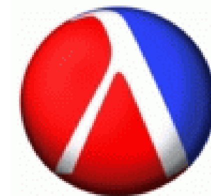


Client
Queue
BadO

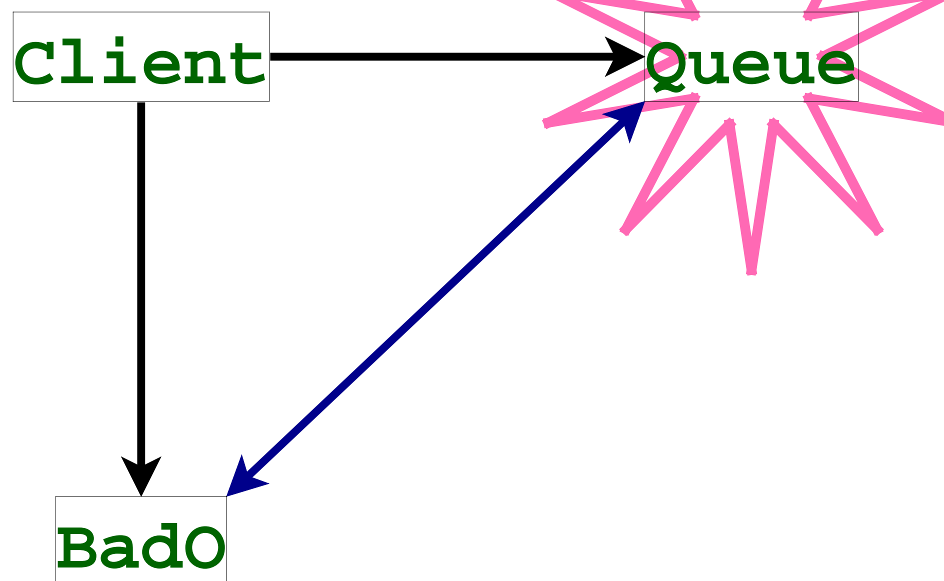


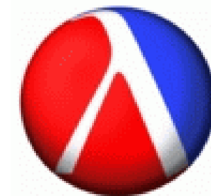
Client links BadO and Queue



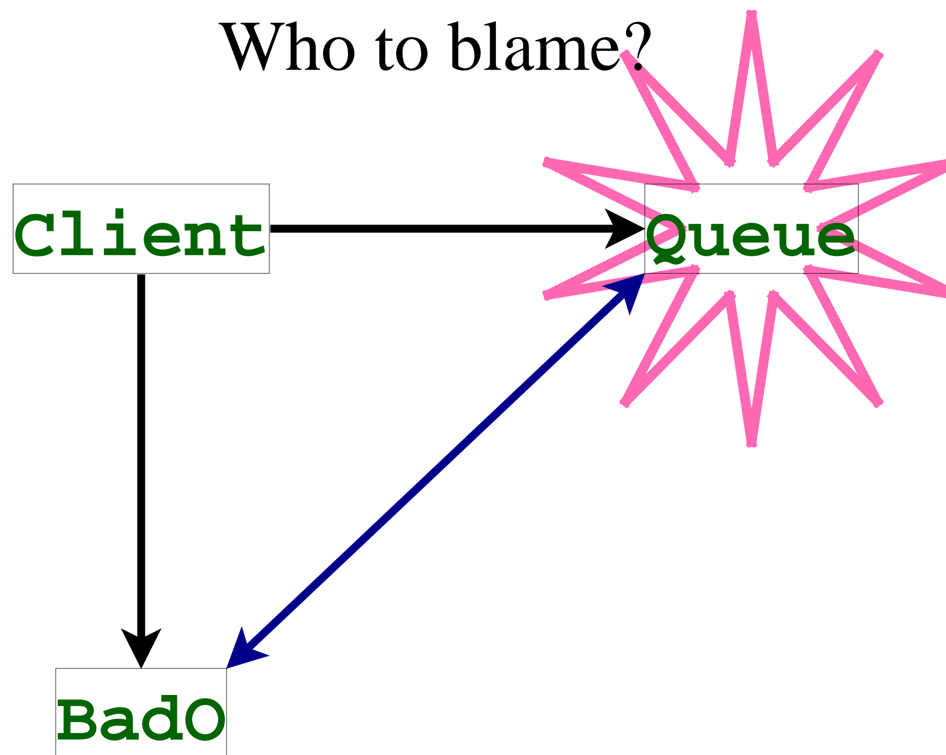


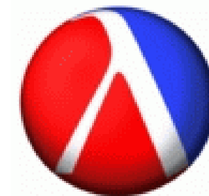
Queue post-condition failure



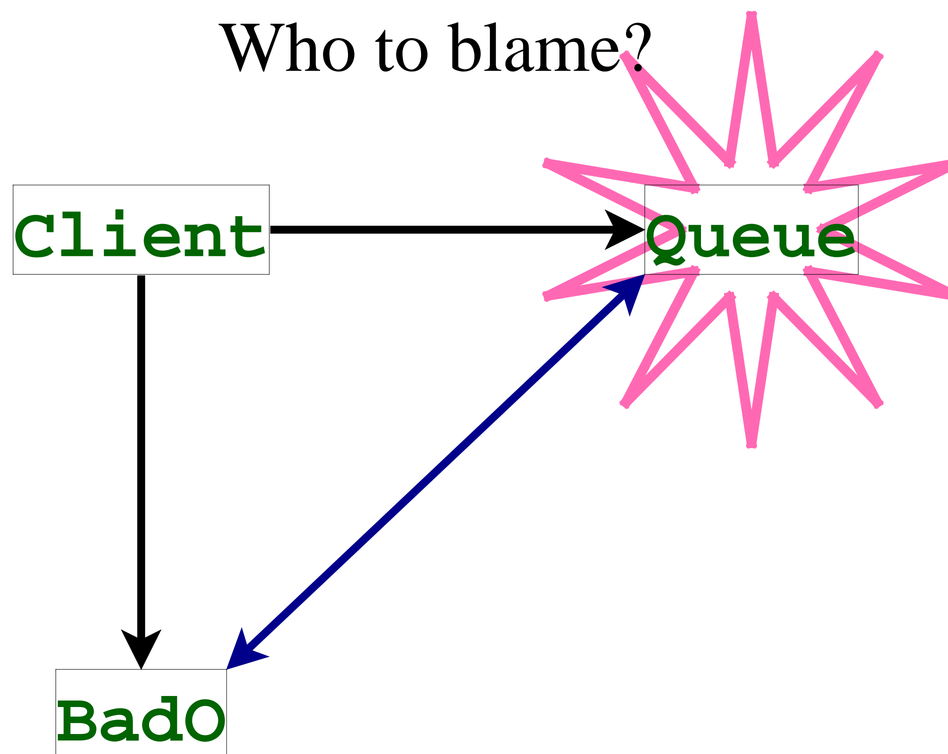


Who to blame?

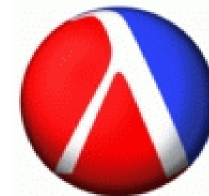




Who to blame?

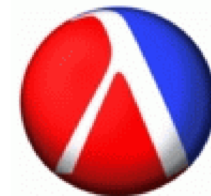


- Client combines mis-matched components
- BadO violates informal contract
- Queue is blamed



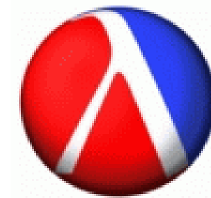
Queue with observer

```
class Q implements IQueue {
  Obs o;

  void enq(int X) {...}
  // @post !this.empty()
  // effect: o.onEnq(this)

  int deq() {...}
  // @pre !this.empty()
  // effect: o.onDeq(this)

  void register(Obs _o) {o = _o;}
  // please: a "good" Observer
}
```

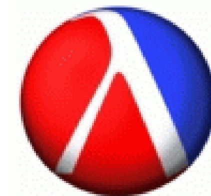


Queue with observer

```
class Q implements IQueue {
  Obs o;

  void enq(int X) {...}
  // @post !this.empty()
  // effect: o.onEnq(this)

  int deq() {...}
  // @pre !this.empty()
  // effect: o.onDeq(this)

  void register(Obs _o) {o = _o;}
  //     @pre _o.onEnq(...)
}
```



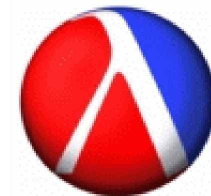
Contracts in interfaces?



Observer contracts

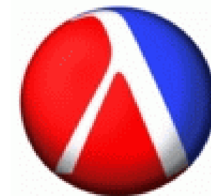
```
interface Obs {
  void init();

  void onEnq(IQueue q);
  // @post !q.empty()

  void onDeq(IQueue q);
  // @pre !q.empty()
}
```

Force observers to meet pre- and post-conditions that Queue needs



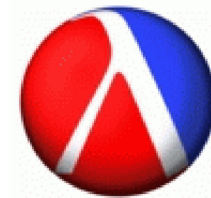
Controlling BadO

```
class BadO
  implements Obs {
  void init() {...}

  void onEnq(IQueue q)
    { q.deq() }


  void onDeq(IQueue q)
    {...}
}
```



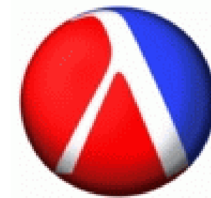
Controlling BadO

```
class BadO
  implements Obs {
  void init() {...}

  void onEnq(IQueue q)
    { q.deq() }
  // @post !q.empty()

  void onDeq(IQueue q)
    {...}
}
```



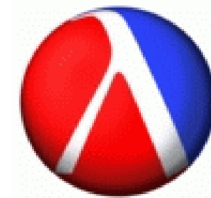
A

A'



A

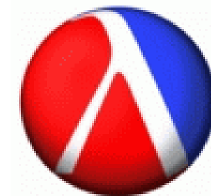
A'



A

A'

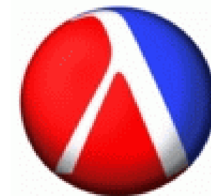
A - A'



A

A'

A - A'

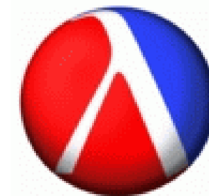


Queue Class

```
class Q implements IQueue {
 ... }
```

Positive Queue

```
interface IPosQ {
  void enq(int X) {...}
  // @pre X >= 0
  // @post !this.empty()

  int deq() {...}
  // @pre !this.empty()
  // @post @ret >= 0
}
```

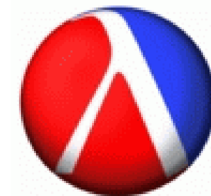


Queue Class

```
class Q implements IQueue {
 ... }
```

Positive Queue

```
interface IPosQ {
  void enq(int X) {...}
  // @pre X >= 0
  // @post !this.empty()

  int deq() {...}
  // @pre !this.empty()
  // @post @ret >= 0
}
```

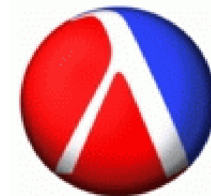


Nominal subtyping

- Hierarchy explicit
- Conventional OO PLs: C++, C#, Eiffel, Java

Structural subtyping

- Hierarchy implicit
- Research OO PLs: Moby, OML, OCaml, LOOM, PolyTOIL

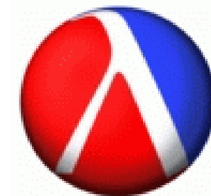


Nominal subtyping

- Simple to implement
- Simple type-error messages
- Inhibits re-use

Structural subtyping

- Harder to implement
- Complex type-error messages
- Permits flexible re-use



QClass

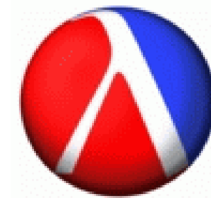
```
class Q implements IQueue {
 ... }
```

IQueue

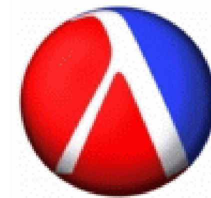
```
interface IQueue {
  void enq(int X) {...}
  // @post !this.empty()


  int deq() {...}
  // @pre !this.empty()

}
```

IPosQ

```
interface IPosQ {
  void enq(int X) {...}
  // @pre X >= 0
  // @post !this.empty()

  int deq() {...}
  // @pre !this.empty()
  // @post @ret >= 0
}
```

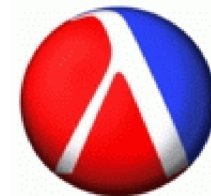


Structural Subtyping in an Nominal World



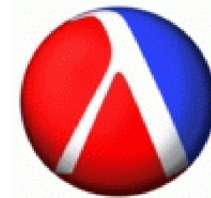
```
semanticCast(obj, I, <fromStr>, <toStr>)
```

A structural subtype "cast"



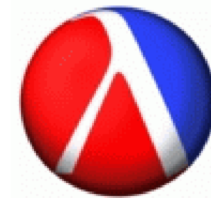
```
semanticCast(obj, I, <fromStr>, <toStr>)
```

The object that gets casted, now has additional contracts



```
semanticCast(obj, I, <fromStr>, <toStr>)
```

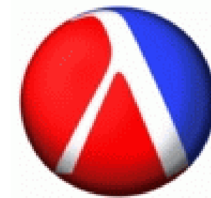
The interface that describes the additional contracts



```
semanticCast(obj, I, <fromStr>, <toStr>)
```

The name of the component
where the object is from;

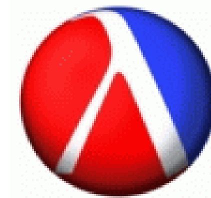
Responsible for post-conds



semanticCast(obj, I, <fromStr>, <toStr>)

The name of the component
where the object is sent;

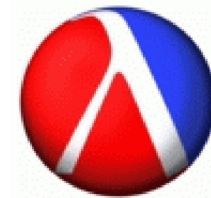
Responsible for pre-conds



```
semanticCast(obj, I, <fromStr>, <toStr>)
```

Result ensures I's contracts
but otherwise identical to obj

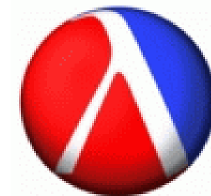
Has type I, even if obj doesn't



A

A'

A - A'



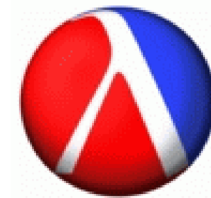
A

A'

semanticCast

(○ , A , ,)

A - A'



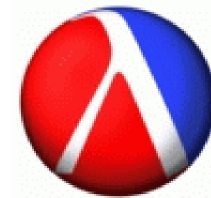
A

A'

semanticCast
(○,A, ,)

semanticCast
(○,A', ,)

A - A'



<Queue>

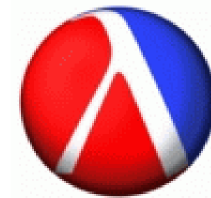
```
IQueue q = new Q();
```

<Client>

```
q.enq(1);
q.deq();
q.enq(-1);
```

<PosQueue>

```
q
```



<Queue>

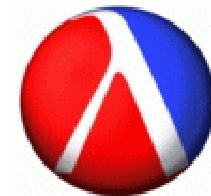
```
IQueue q = new Q();
```

<Client>

```
q.enq(1);
q.deq();
q.enq(-1);
```

<PosQueue>

```
q
```



<Queue>

```
IQueue q = new Q();
```

<Client>

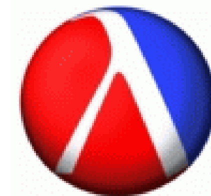
```
q.enq(1);
q.deq();
q.enq(-1);
```

```
semanticCast(q,
             IQueue,
             <Queue>,
             <PosQueue>)
```

```
semanticCast(q,
             IPosQ,
             <PosQueue>,
             <Client>)
```

<PosQueue>

```
q
```



<Queue>

```
IQueue q = new Q();
```

<Client>

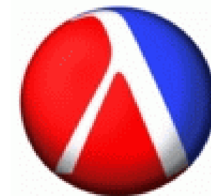
```
q.enq(1);
q.deq();
q.enq(-1);
```

```
semanticCast(q,
             IQueue,
             <Queue>,
             <PosQueue>)
```

```
semanticCast(q,
             IPosQ,
             <PosQueue>,
             <Client>)
```

<PosQueue>

```
q
```



<Queue>

```
IQueue q = new Q();
```

<Client>

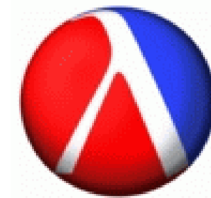
```
q.enq(1);
q.deq();
q.enq(-1);
```

```
semanticCast(q,
             IQueue,
             <Queue>,
             <PosQueue>)
```

```
semanticCast(q,
             IPosQ,
             <PosQueue>,
             <Client>)
```

<PosQueue>

```
q
```



<Queue>

```
IQueue q = new Q();
```

<Client>

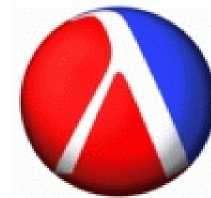
```
q.enq(1);
q.deq();
q.enq(-1);
```

```
semanticCast(q,
             IQueue,
             <Queue>,
             <PosQueue>)
```

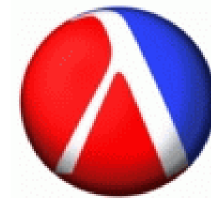
```
semanticCast(q,
             IPosQ,
             <PosQueue>,
             <Client>)
```

<PosQueue>

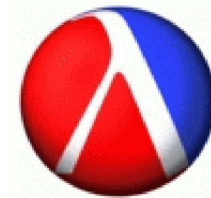
```
q
```



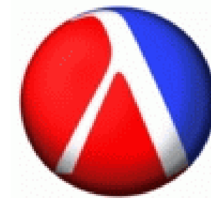
Semantics of semanticCast



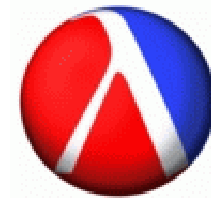
Semantics of semanticCast



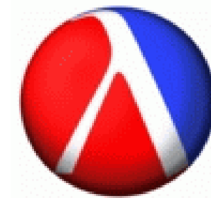
Semantics of semanticCast



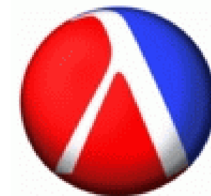
Semantics of semanticCast



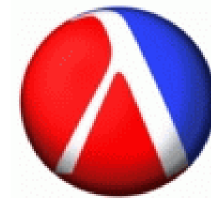
Semantics of semanticCast



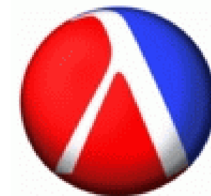
Semantics of semanticCast



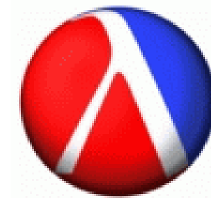
Semantics of semanticCast



Semantics of semanticCast

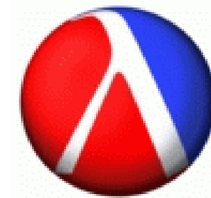


```
semanticCast(o, I, <from>, <to>).m(o')
=
semanticCast(o.m(semanticCast o', J, <to>, <from>),
             K,
             <from>,
             <to>)

interface I { K m(J x); }
interface J { ... }
interface K { ... }
```

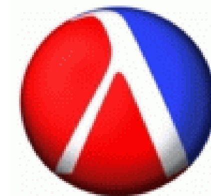


Implementation

- Proxies
- Construct new proxies at method calls

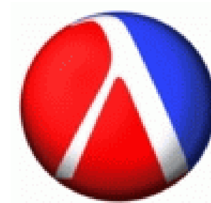


Wrap up



Structural subtyping for contracts

- Adds flexibility to conventional languages
- Contracts still simple boolean expressions
- Proper blame assignment
- If a tree in the forest hasn't yet fallen,
  it didn't make a sound



Thank you.