SOFTWARE HARDENING

Sylvain Lebresne, **Johan Ostlund**, **Greg Richards**, ***Jan Vitek***, **Tobias Wrigstad**, *Purdue University*
Francesco Zappa Nardelli, *INRIA*
Bard Bloom, John Field, **Nate Nystrom**, *IBM Research*
Rok Strnisa, *Cambridge University*

SCRIPTING LANGUAGES

- … lightweight, dynamic languages designed to maximize productivity by offering high-level abstractions and reducing the syntactic overhead found in most programming languages.
- exempli gatia:
  Perl, Python, Tcl, Ruby, JavaScript, Groovy, Smalltalk, Scheme…

MOTIVATION: PLUTO

- Pluto or *Premiepensionmyndigheten* (PPM)
- 5.5 million users
- Managing 250 000 000 000 SEK (23 billion euros)
- Up to 30 developers for 7 years
- 750 gigabytes of data in an Oracle database
- **320 000 lines of…**

  Perl

MOTIVATION

- Advantages of scripting languages include:
  - Permissive - *dynamic languages are maximally permissive, anything goes, until it doesn't.*
  - Modular - *dynamic languages are maximally modular, a program can be run even when crucial pieces are missing*
- These features enable very rapid development of software

MOTIVATION

- Drawbacks of "Scripting" over "programming" languages:
  - Performance
  - Errors are caught at runtime
  - No (good) concurrency story

MOTIVATION

- Software Hardening refers to the a tradeoff between flexibility and assurance or performance
- We propose to investigate three dimensions
  - Incremental Type Hardening
  - Incremental Contract Hardening
  - Incremental Data Hardening

THE THORN PROJECT

- Joint project with IBM Research
- Its scientific goals are to find programming language techniques to facilitate the incremental transition from scripts to programs™
- The vehicle for our work is an experimental platform for language research called Thorn
- Thorn is a new language designed to support incremental software hardening of untyped, dynamic code

REQUIREMENTS

- When designing a language it is helpful to have some guiding principles:
  - Permissive - *try to accept as many scripts as possible*
  - Modular - *be as modular as possible*
  - Reward good behavior - *programmer effort rewarded either with performance or clear correctness guarantee*

THORN

BASIC LANGUAGE DESIGN

- Thorn is a class-based object-oriented language
- Thorn does not support
    - field/method deletion
    - field/method addition
    - dynamic class hierarchy changes
        - *Why not scheme, or JavaScript?*

BASIC LANGUAGE DESIGN

- We are looking at how these features are used in JavaScript

	Meth. add.	Meth. upd.	Field add.	Proto. upd.	Deletions	Avg.
	Tot./Obj.	Tot./Obj.	Tot./Obj.	Tot./Obj.	Tot./Obj.	(Med.)
3d-cube	0/0	0/0	16/4	0/0	0/0	4.0 (2)
3d-raytrace	2/2	0/0	124/64	0/0	0/0	1.9 (2)
binary-trees	0/0	0/0	0/0	0/0	0/0	0.0 (0)
v8-crypto	61/4	0/0	950/475	0/0	0/0	2.1 (2)
v8-deltablue	11/8	0/0	10/2	12/12	0/0	2.2 (2)
v8-raytrace	587/77	10/5	180/36	33/33	0/0	6.4 (2)
v8-richards	0/0	0/0	0/0	0/0	0/0	0.0 (0)
amazon	2160/4198	39/67	7050/59769	2/2	1174/1896	8.4 (2)
basecamp	112/819	7/7	142/1883	0/0	0/0	11.6 (2)
facebook	5212/16432	256/648	19787/84912	72/72	352/727	4.3 (2)
gmail	2123/4258	68/180	10982/35783	1896/1896	6001/19972	3.3 (2)
livelykernel	21605/42346	0/0	15555/16584	0/0	0/0	2.6 (2)
nasa	421/2045	361/361	2621/6127	7/7	1/3	2.6 (1)
random	1885/4037	24/1563	6188/48988	121/121	69/173	6.8 (2)

CLASSES

```
class Point(x,y);
```

CLASSES

```
class Point {
   val x;
   val y;
   Point(x’, y’) { x=x’; y=y’; } # constructor
  ~Point(x,y);                   # deconstructor
}
```

MULTIPLE INHERITANCE

```
class Flavor(fl);

class TastyPoint(x,y,fl) extends
                Point(x,y), Flavor(fl);
```

PATTERN MATCHING

```
match(pt) {
    TastyPoint(x,y,f) => “$(x) $(y) $(f)”
  | Point(x,y)        => “$(x) $(y)”
}
```

TYPE HARDENING

SOFT TYPING

Cartwright and Fagan, 1991

```
class Point(x,y) {
  fun move(p) {
    x := p.gteX();
    y := p.getY();
  }
}
```

Rather than rejecting a program that cannot be typed, insert appropriate run-time checks.

GRADUAL TYPING

Siek and Taha 2006, Siek and Taha 2007

- The transition from untyped to typed should happen gradually
- Gradual typing: whenever we go from typed to untyped code, insert the appropriate cast

GRADUAL TYPING

```
class Foo { fun bar(x: Int) x*x; }

f: Foo = Foo();
f.bar(xyzzy); # does not type check
```

Here an implicit cast is inserted at the call-site.

GRADUAL TYPING

```
class Foo { fun bar(x: Int) x*x; }

f: Foo = Foo();
f.bar((Int) xyzzy); # OK
```

Here an implicit cast is inserted at the call-site.

GRADUAL TYPING

```
class Ordered {fun compare(o:Ordered):Int;}
class SubString {fun sub(o:String):Bool;}
fun sort(x: [Ordered]):[Ordered] = …
fun filter(x: [SubString]):[SubString] = …
```

GRADUAL TYPING

```
class Ordered {fun compare(o:Ordered):Int;}
class SubString {fun sub(o:String):Bool;}
fun sort(x: [Ordered]):[Ordered] = …
fun filter(x: [SubString]):[SubString] = …

fun  funny( f: dyn ) {
  f‘:[SubString] = filter(sort(f));
  # f’ = ([SubString])([Ordered])f
  v:SubString = f’[0];
  # v =  (Substring)(Ordered)f[0]
```

THE TYPING OF POINT

THE TYPING OF A POINT

```
class Point(var x, var y) {
  fun getX() = x;
  fun getY() = y;
  fun move(p) { x:=p.getX(); y:=p.getY()}
}

o = Point(0,0);   # create a point
a = Point(5,6);   # create another point
a.move(o);        # move point a to point o
```

THE TYPING OF A POINT

```
class Point(var x: Int, var y: Int) {
   fun getX(): Int = x;
   fun getY(): Int = y;
   fun move(p:Point) {
       x := p.getX(); y := p.getY()
   }
}
```

THE TYPING OF A POINT

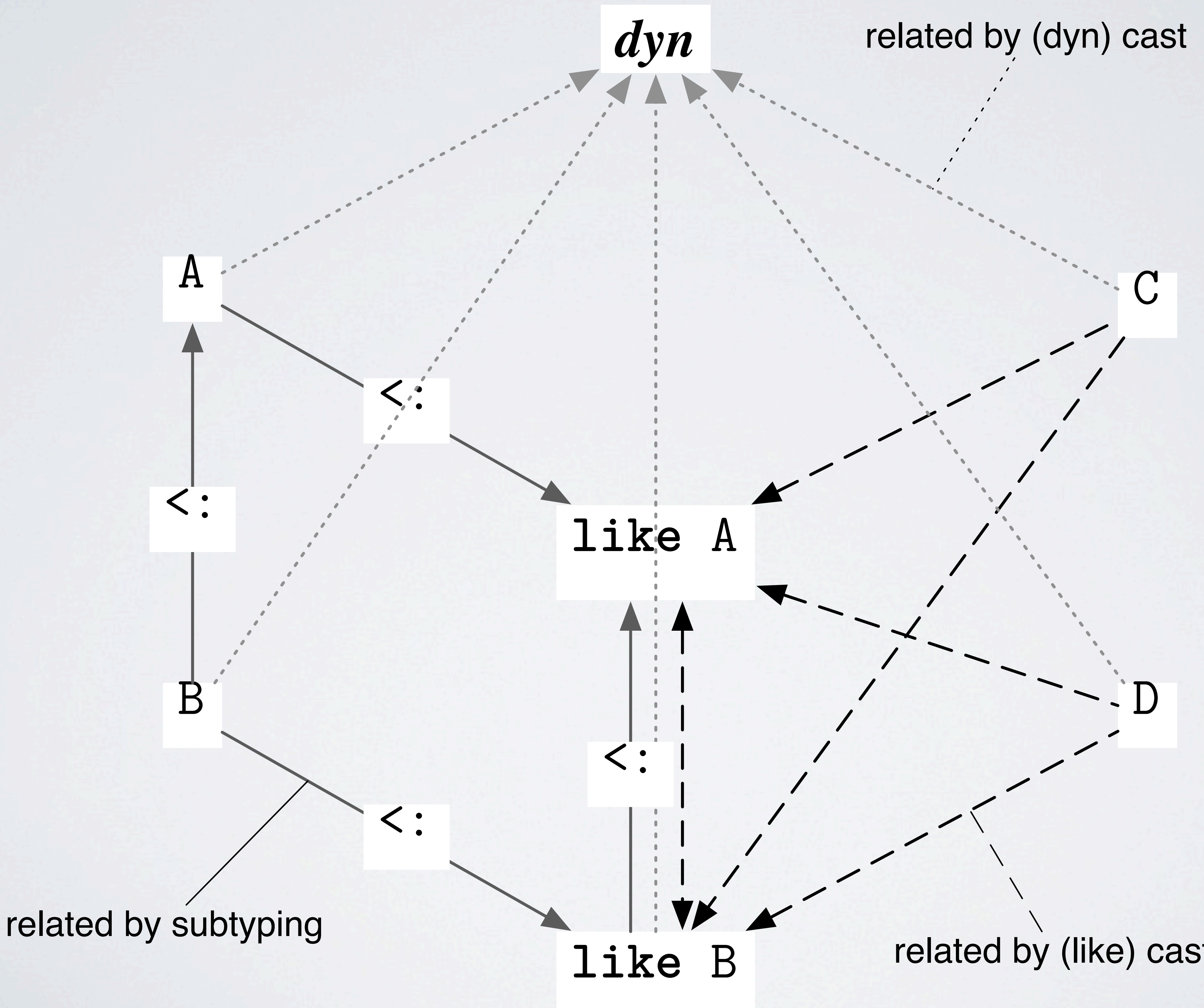
```
class Coordinate(var x: Int, var y: Int) {
  fun getX(): Int = x;
  fun getY(): Int = y;
}

p = Point(0,0);
c = Coordinate(5,6);
p.move(c);
```

LIKE TYPES

```
fun move(p:like Point) {
  x := p.getX();
  y := p.getY();
  # p.hog();      # raises compile-time err
}
```

LIKE TYPES



dyn
related by (dyn) cast
A
C
<:
<:
like A
B
D
<:
<:
related by subtyping
like B
related by (like) cast

INTERFACING TYPED AND UNTYPED CODE

```
class Point{...fun move(p:Point)...}

fun moveIt(p1, p2: Point, p3: like Point) {
  p1.move(p3);
  p2.move(p2);
  p3.move(p1);
}
```

LIKE TYPES

- A unilateral promise as to how a value will be treated locally
- Allows most of the regular static checking machinery
- Allows the flexibility of structural subtyping at a lower cost
- Concrete types can stay concrete so more aggressive optimisations are possible
- Reusing type names as semantic tags

CONCLUSION

- Like types represent a sweet spot in the design space of language features for incremental hardening of software
- Still not enough experience to draw strong conclusions
- Contracts capture richer semantic properties than types but are (usually) more expensive to check at runtime. Are there other sweet spots?