EVOLUTION FROM SCRIPTS TO PROGRAMS: A SURVEY

Tobias Wrigstad
Stockholm University

onsdag den 11 november 2009
WHAT ARE THE **PROS** OF SCRIPTING/DYNAMIC PL$$
Simple
Simple

Flexible
Simple

Flexible

Pragmatic
Simple
Flexible
Pragmatic
Low coupling
Simple
Flexible
Pragmatic
Malleable
Low coupling
Simple

Flexible

Few commitments

Malleable

Pragmatic

Low coupling
Simple
Flexible
Pragmatic
Few commitments
Malleable
Go well with Agile et al.
Low coupling
Simple
Flexible
Pragmatic
Malleable
Few commitments
Low coupling
Go well with Agile et al.
Fun?
Simple
Flexible
Pragmatic
Malleable
Few commitments
Unobtrusive
Go well with Agile et al.
Low coupling
Fun?
Simple
Flexible
Pragmatic
Malleable
Low coupling
Few commitments
High-level
Go well with Agile et al.
Unobtrusive
Fun?
Simple
Flexible
Pragmatic
Malleable
Low coupling
Few commitments
High-level
Go well with Agile et al.
Succinct
Fun?
Unobtrusive
WHAT ARE THE CONS?

onsdag den 11 november 2009
Slow
Slow

Too pragmatic?
Slow

Too pragmatic?

No static safety
Slow

Too pragmatic?

Hard to do program analysis

No static safety
Slow

Too pragmatic?

Hard to maintain

Hard to do program analysis

No static safety
Slow

Too pragmatic?

Hard to maintain

Hard to do program analysis

Optimisation is tricky

No static safety
Slow

Too pragmatic?

Hard to maintain

Hard to do program analysis

Unreadable

Optimisation is tricky

No static safety
Slow

Too pragmatic?

Hard to maintain

Hard to do program analysis

Optimisation is tricky

No static safety

Unreadable

Bad IDE support
Slow

Too pragmatic?

Unreadable

Hard to maintain

Hard to do program analysis

No encapsulation

Optimisation is tricky

No static safety

Bad IDE support

onsdag den 11 november 2009
BEST OF BOTH WORLDS

• As programs mature and stabilise we would like to be able to evolve them from **scripts** to **programs**

• Script = untyped, flexible code

• Program = typed, more rigid code
PROBLEM I: SOME THINGS ARE INTRINSICALLY HARD TO TYPE

```kotlin
fun foo(a, b) {
    if (a) {
        // treat b as a String
    } else {
        // treat b as an Int
    }
}
```
PROBLEM II: TYPE INFRINGEMENT

class Int {
    fun +(term: Int) { ... }
}

fun foo(a) {
    a+1; # OK
    1+a; # NOT OK
}

onsdag den 11 november 2009
fun toto(x: Quux): Bar { ... }

import toto;

var x = ...;
var y = toto(x);  # x’s type + return
               # type lost
WHAT KIND OF TYPES SHOULD WE USE?
NOMINAL TYPING

class Foo { fun bar() = 23; }
class Bar { fun bar() = 23; }

var x: Foo = Bar(); # works?

A type is named entity. Type equivalence is name equivalence. Subtyping must be explicitly declared.
A type is defined by a structure. Identical structures have the same type. Subtyping is superset (almost).
UNION TYPES

Igarashi & Nagira 2007 (Cardone and Dezani-Ciancaglini, 1994)

```plaintext
class A { fun f() = A(); }
class B { fun f() = B(); }

A ∪ B

If we take a set interpretation of types, then [[A]] is the set of all objects that A describes. What is [[A ∪ B]]?
```
A WEALTH OF TYPES

• Dependent types (types dependent on run-time values)
  A lot of people. See e.g., Altenkirch et al. 2005.

• Intersection types (overlapping intersection between types)
  Cardone and Dezani-Ciancaglini, 1994

• Session types (type communication exchanges)
  Coppo et al. 2005, Giunti et al. 2008

• Type states (roughly state machines encoded in the types)
  Strom and Yellin 1986, DeLine and Fähndrich 2002, others…

• Ownership types (encode nesting structure in objects)

• …
WHAT **STRATEGY** FOR TYPING SHOULD WE USE?
By examining a method body, we can infer constraints which arguments must satisfy for the method to work. What are the pros and cons?
class Point(x,y) {
    fun move(p) {
        x := p.gteX();
        y := p.getY();
    }
}

Rather than rejecting a program that cannot be typed (e.g., for lack of type information), we insert appropriate run-time checks. Can anyone spot a problem?
SOFTPHP
Camphuijsen, Hage & Holdermans 2009

- Constraint-based analysis for subset of PHP (notably not classes and objects)
  - Warns and weights ”seriousness of error”
- Most functions do not propagate constraints
- Quite a few warnings given in working programs, less warnings in mature code (gives some confidence)
HYBRID TYPING
Flanagan, 2006

Static typing is (generally) imprecise (\texttt{int age = -27;})

• Refinement types help here
  • Non-null types, intervals, valid array index, etc.

• \textbf{Downside:} hard to program with
  • Hybrid typing inserts run-time checks where compile-time proofs cannot be constructed
GRADUAL TYPING

Siek and Taha 2006, Siek and Taha 2007

• The transition from untyped to typed should happen gradually

• Naïve approaches will potentially require an entire program to become typed as result of inserting some type information in some locations

• Gradual typing: whenever we go from typed to untyped code, insert the appropriate cast
class Foo { fun bar(x: Int) x*x; }

f: Foo = Foo();
f.bar(\texttt{xyzzy}); # does not type check

Here an implicit cast is inserted at the call-site. Can anyone spot a problem with efficient implementation?
GRADUAL TYPING

Siek and Taha 2006, Siek and Taha 2007

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. Can anyone spot a problem with efficient implementation?
STRONGTALK & TYPEPLUG

Bracha and Griswold 1993

Haldiman et al. 2009

• Annotate Smalltalk code with **optional** type annotations

• Optional and **extensible** type system for Smalltalk
  • Example: non-null types
  • Type inference as an aid—not for rejecting programs

• Support for union types, parametric polymorphism, structural typing

• No optimisation, but type checking for typed parts of code
STRONGTALK & TYPEPLUG

Bracha and Griswold 1993

Annotate Smalltalk code with optional type annotations

Support for union types, parametric polymorphism, structural typing

No optimisation, but type checking for typed parts of code

Optional and extensible type system for Smalltalk

Type inference as an aid, not for rejecting programs

Beatrice’s topic for next Thursday!

Bracha and Griswold 1993

Haldiman et al. 2009
class Frob { ...23 methods... }

fun foo(arg: Frob) {
    ...use 2 methods in Frob...
}

Even if typing is structural, it is not likely to be very flexible in practise here (why?). Can anyone see a solution?
IS STRUCTURAL TYPING USEFUL?

Malayeri and Aldrich, 2009

• Recurring structural types in Java collections libraries (e.g.,
  \{ hasNext(), next() \})

• Some degree of name reuse in Java
Malayeri and Aldrich, 2009, cont’d

<table>
<thead>
<tr>
<th>Application</th>
<th>LOC</th>
<th>Number of types</th>
<th>Types with &gt;1 common method</th>
<th>Percentage</th>
<th>% common methods</th>
<th>Avg # classes/common signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>JFreeChart</td>
<td>93k</td>
<td>789</td>
<td>301</td>
<td>38.1%</td>
<td>39.5%</td>
<td>3.9</td>
</tr>
<tr>
<td>Struts</td>
<td>28k</td>
<td>609</td>
<td>158</td>
<td>25.9%</td>
<td>45.2%</td>
<td>2.7</td>
</tr>
<tr>
<td>quartz</td>
<td>22k</td>
<td>158</td>
<td>24</td>
<td>15.2%</td>
<td>20.0%</td>
<td>2.4</td>
</tr>
<tr>
<td>Smack</td>
<td>40k</td>
<td>847</td>
<td>115</td>
<td>13.6%</td>
<td>23.5%</td>
<td>3.3</td>
</tr>
<tr>
<td>Tomcat</td>
<td>126k</td>
<td>1727</td>
<td>234</td>
<td>13.5%</td>
<td>32.6%</td>
<td>3.6</td>
</tr>
<tr>
<td>antlr</td>
<td>42k</td>
<td>226</td>
<td>26</td>
<td>11.5%</td>
<td>23.6%</td>
<td>2.7</td>
</tr>
<tr>
<td>poi</td>
<td>50k</td>
<td>539</td>
<td>62</td>
<td>11.5%</td>
<td>20.9%</td>
<td>2.6</td>
</tr>
<tr>
<td>OpenFire</td>
<td>90k</td>
<td>1039</td>
<td>110</td>
<td>10.6%</td>
<td>19.0%</td>
<td>3.0</td>
</tr>
<tr>
<td>freecol</td>
<td>62k</td>
<td>569</td>
<td>55</td>
<td>9.7%</td>
<td>20.6%</td>
<td>2.7</td>
</tr>
<tr>
<td>JHotDraw</td>
<td>52k</td>
<td>616</td>
<td>59</td>
<td>9.6%</td>
<td>19.0%</td>
<td>2.8</td>
</tr>
<tr>
<td>hsqldb</td>
<td>62k</td>
<td>355</td>
<td>31</td>
<td>8.7%</td>
<td>19.5%</td>
<td>2.6</td>
</tr>
<tr>
<td>jruby</td>
<td>86k</td>
<td>997</td>
<td>83</td>
<td>8.3%</td>
<td>15.6%</td>
<td>3.1</td>
</tr>
<tr>
<td>Areca</td>
<td>35k</td>
<td>362</td>
<td>30</td>
<td>8.3%</td>
<td>15.4%</td>
<td>2.7</td>
</tr>
<tr>
<td>HttpClient</td>
<td>18k</td>
<td>231</td>
<td>19</td>
<td>8.2%</td>
<td>15.0%</td>
<td>2.6</td>
</tr>
<tr>
<td>xalan</td>
<td>161k</td>
<td>1223</td>
<td>94</td>
<td>7.7%</td>
<td>16.1%</td>
<td>2.9</td>
</tr>
<tr>
<td>plt collections</td>
<td>19k</td>
<td>812</td>
<td>60</td>
<td>7.4%</td>
<td>7.5%</td>
<td>2.8</td>
</tr>
<tr>
<td>Cayenne</td>
<td>95k</td>
<td>1415</td>
<td>104</td>
<td>7.3%</td>
<td>18.1%</td>
<td>2.8</td>
</tr>
<tr>
<td>DrJava</td>
<td>59k</td>
<td>927</td>
<td>65</td>
<td>7.0%</td>
<td>12.1%</td>
<td>2.6</td>
</tr>
<tr>
<td>Ant</td>
<td>62k</td>
<td>945</td>
<td>65</td>
<td>6.9%</td>
<td>31.3%</td>
<td>3.7</td>
</tr>
<tr>
<td>Lucene</td>
<td>24k</td>
<td>398</td>
<td>21</td>
<td>5.3%</td>
<td>13.4%</td>
<td>2.6</td>
</tr>
<tr>
<td>LimeWire</td>
<td>97k</td>
<td>1689</td>
<td>88</td>
<td>5.2%</td>
<td>17.7%</td>
<td>3.1</td>
</tr>
<tr>
<td>pmd</td>
<td>38k</td>
<td>478</td>
<td>24</td>
<td>5.0%</td>
<td>12.0%</td>
<td>2.7</td>
</tr>
<tr>
<td>Emma</td>
<td>23k</td>
<td>443</td>
<td>22</td>
<td>5.0%</td>
<td>18.7%</td>
<td>3.4</td>
</tr>
<tr>
<td>jEdit</td>
<td>71k</td>
<td>880</td>
<td>40</td>
<td>4.5%</td>
<td>11.7%</td>
<td>2.5</td>
</tr>
<tr>
<td>jung</td>
<td>26k</td>
<td>531</td>
<td>24</td>
<td>4.5%</td>
<td>19.3%</td>
<td>2.4</td>
</tr>
<tr>
<td>Columba</td>
<td>70k</td>
<td>1232</td>
<td>48</td>
<td>3.9%</td>
<td>17.3%</td>
<td>3.1</td>
</tr>
<tr>
<td>Apache Collections</td>
<td>26k</td>
<td>550</td>
<td>19</td>
<td>3.5%</td>
<td>7.3%</td>
<td>2.7</td>
</tr>
<tr>
<td>Crystal</td>
<td>12k</td>
<td>211</td>
<td>4</td>
<td>1.9%</td>
<td>5.1%</td>
<td>2.9</td>
</tr>
</tbody>
</table>

Average: 15.9% 26.4% 2.9
Malayeri and Aldrich, 2009, cont’d

<table>
<thead>
<tr>
<th>Application</th>
<th>LOC</th>
<th>Number of types</th>
<th>Types with &gt;1 common method</th>
<th>Percentage</th>
<th>% common methods</th>
<th>Avg # classes/ common signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>JFreeChart</td>
<td>93k</td>
<td>789</td>
<td>301</td>
<td>38.1%</td>
<td>39.5%</td>
<td>3.9</td>
</tr>
<tr>
<td>Struts</td>
<td>28k</td>
<td>609</td>
<td>158</td>
<td>25.9%</td>
<td>45.2%</td>
<td>2.7</td>
</tr>
<tr>
<td>quartz</td>
<td>22k</td>
<td>158</td>
<td>24</td>
<td>15.2%</td>
<td>20.0%</td>
<td>2.4</td>
</tr>
<tr>
<td>Smack</td>
<td>40k</td>
<td>847</td>
<td>115</td>
<td>13.6%</td>
<td>23.5%</td>
<td>3.3</td>
</tr>
<tr>
<td>Tomcat</td>
<td>126k</td>
<td>1727</td>
<td>234</td>
<td>13.5%</td>
<td>32.6%</td>
<td>3.6</td>
</tr>
<tr>
<td>antlr</td>
<td>42k</td>
<td>226</td>
<td>26</td>
<td>11.5%</td>
<td>23.6%</td>
<td>2.7</td>
</tr>
<tr>
<td>poi</td>
<td>50k</td>
<td>539</td>
<td>62</td>
<td>11.5%</td>
<td>20.9%</td>
<td>2.6</td>
</tr>
<tr>
<td>OpenFire</td>
<td>90k</td>
<td>1039</td>
<td>110</td>
<td>10.6%</td>
<td>19.0%</td>
<td>3.0</td>
</tr>
<tr>
<td>freecol</td>
<td>62k</td>
<td>569</td>
<td>55</td>
<td>9.7%</td>
<td>20.6%</td>
<td>2.7</td>
</tr>
<tr>
<td>JHotDraw</td>
<td>52k</td>
<td>616</td>
<td>59</td>
<td>9.6%</td>
<td>19.0%</td>
<td>2.8</td>
</tr>
<tr>
<td>hsqldb</td>
<td>62k</td>
<td>355</td>
<td>31</td>
<td>8.7%</td>
<td>19.5%</td>
<td>2.6</td>
</tr>
<tr>
<td>jruby</td>
<td>86k</td>
<td>997</td>
<td>25</td>
<td>8.2%</td>
<td>15.0%</td>
<td>2.1</td>
</tr>
<tr>
<td>Areca</td>
<td>35k</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HttpClient</td>
<td>18k</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xalan</td>
<td>161k</td>
<td>1223</td>
<td>94</td>
<td>7.1%</td>
<td>16.1%</td>
<td>2.9</td>
</tr>
<tr>
<td>plt collections</td>
<td>19k</td>
<td>812</td>
<td>60</td>
<td>7.4%</td>
<td>7.5%</td>
<td>2.8</td>
</tr>
<tr>
<td>Cayenne</td>
<td>95k</td>
<td>1415</td>
<td>104</td>
<td>7.3%</td>
<td>18.1%</td>
<td>2.8</td>
</tr>
<tr>
<td>DrJava</td>
<td>59k</td>
<td>927</td>
<td>65</td>
<td>7.0%</td>
<td>12.1%</td>
<td>2.6</td>
</tr>
<tr>
<td>Ant</td>
<td>62k</td>
<td>945</td>
<td>65</td>
<td>6.9%</td>
<td>31.3%</td>
<td>3.7</td>
</tr>
<tr>
<td>Lucene</td>
<td>24k</td>
<td>398</td>
<td>21</td>
<td>5.3%</td>
<td>13.4%</td>
<td>2.6</td>
</tr>
<tr>
<td>LimeWire</td>
<td>97k</td>
<td>1689</td>
<td>88</td>
<td>5.2%</td>
<td>17.7%</td>
<td>3.1</td>
</tr>
<tr>
<td>pmd</td>
<td>38k</td>
<td>478</td>
<td>24</td>
<td>5.0%</td>
<td>12.0%</td>
<td>2.7</td>
</tr>
<tr>
<td>Emma</td>
<td>23k</td>
<td>443</td>
<td>22</td>
<td>5.0%</td>
<td>18.7%</td>
<td>3.4</td>
</tr>
<tr>
<td>jEdit</td>
<td>71k</td>
<td>880</td>
<td>40</td>
<td>4.5%</td>
<td>11.7%</td>
<td>2.5</td>
</tr>
<tr>
<td>jung</td>
<td>26k</td>
<td>531</td>
<td>24</td>
<td>4.5%</td>
<td>19.3%</td>
<td>2.4</td>
</tr>
<tr>
<td>Columba</td>
<td>70k</td>
<td>1232</td>
<td>48</td>
<td>3.9%</td>
<td>17.3%</td>
<td>3.1</td>
</tr>
<tr>
<td>Apache Collections</td>
<td>26k</td>
<td>550</td>
<td>19</td>
<td>3.5%</td>
<td>7.3%</td>
<td>2.7</td>
</tr>
<tr>
<td>Crystal</td>
<td>12k</td>
<td>211</td>
<td>4</td>
<td>1.9%</td>
<td>5.1%</td>
<td>2.9</td>
</tr>
</tbody>
</table>

Average 15.9% 26.4% 2.9
LIKE TYPES
Wrigstad, Zappa Nardelli, Lebresne. Östlund and Vitek 2009

• 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
class Point(x: Int, y: Int) {
    fun move(p: like Point) {
        x := p.x; y := p.y;
    }
}

Argument to move should behave "like a point."
Can be assigned from wherever. May not be entirely point-like all. But inside of move is statically typed.
fun a(i, j) =
    1.0 / (((i + j) * (i + j + 1) >> 1) + i + 1);

87 bytecode instructions, 8 new frames, 8 new objects…
fun a(i: Int32, j: Int32) =
    1.0 / (((i + j) * (i + j + 1) >> 1) + i + 1);

29 bytecode instructions, 0 new frames, 1 new object (because of untyped return)
TYPES FOR OPTIMISING THORN
Wrigstad, Östlund and Nystrom

shootout.alioth.debian.org/

onsdag den 11 november 2009
FUTURE: TYPING JAVASCRIPT
Lebresne, Wrigstad, Östlund, Richards, Vitek, 2009—

• What is a suitable definition of a type?
  • Structural? Nominal? Unions? Parametric polymorphism?

• How do programmers program in JavaScript? What typing disciplines arise?

• Can unsafe practises be recoded in a more-safe manner?
HOW TYPE THIS?
(posed by Vitek)

```javascript
function foo(x) {
  if (x)
    this.f = function ...
    this.g = 42;
}
```

Depending on the value of `x`, objects created from `foo` will sometimes have a function `f` or not. Problem?
WHAT IS GOOD ABOUT DYNAMIC TYPING?

• Studies [e.g., Holkner and Harland 2009] show that programs written in dynamic programming languages aren’t really that dynamic.

• What does this suggest about dynamic typing?
CONCLUSIONS I

• Languages in the dynamic/scripting family are good for speedy development and a good fit for modern approaches to software development

• Maturing still means rewriting in another language, mostly

• The academic PL community is now interested in scripting and untyped languages again

• Gradual typing is enjoying being hip (and added to Python)

By Siek and his student
CONCLUSIONS II

• The question of how to best balance flexibility and safety remains an open one

• Gradual typing and similar approaches are maturing, will soon see widespread adoption—AIR! ECMAScript4 is dead :-(

• Like types seem like a good fit for dynamic languages—give local guarantees and but impose no requirements on clients

• More research is needed to find how "soft" the checking should be
QUESTIONS?