Beyond C, C++, Python and Perl

André Pang

ozone@algorithm.com.au

June 25, 2004

README

This is not a language advocacy talk:

- Every language has (dis)advantages
- Use what's best for the task-at-hand
- Multi-language approach?

This is an awareness talk:

- … on programming language *research*
- Not necessarily implying to use this in production
- High-level introduction, not low-level implementation details

README

Don’t hear things I’m not saying!

- *I discussed the typing system in Standard ML … Part way through the explanation, one of the audience members raised his hand and asked “But what's wrong with the way Perl does it?”* — Mark Jason-Dominus

Strong vs Weak Typing

Strong vs weak:

- How easy/necessary is it to violate the type system?
- Weak type systems = easy to violate, or require it as part of typical programming:

```
void callback_function (void *data) {
  my_type *data = (my_type *) data;
}
```

Strong typing is always better:

- Strongly typed languages don't allow things to 'go wrong' (no segfaults, no out-of-bounds errors, no buffer overflows)
- Some strongly typed languages are mathematically proven (Haskell 98, Standard ML)

Static vs Dynamic Typing

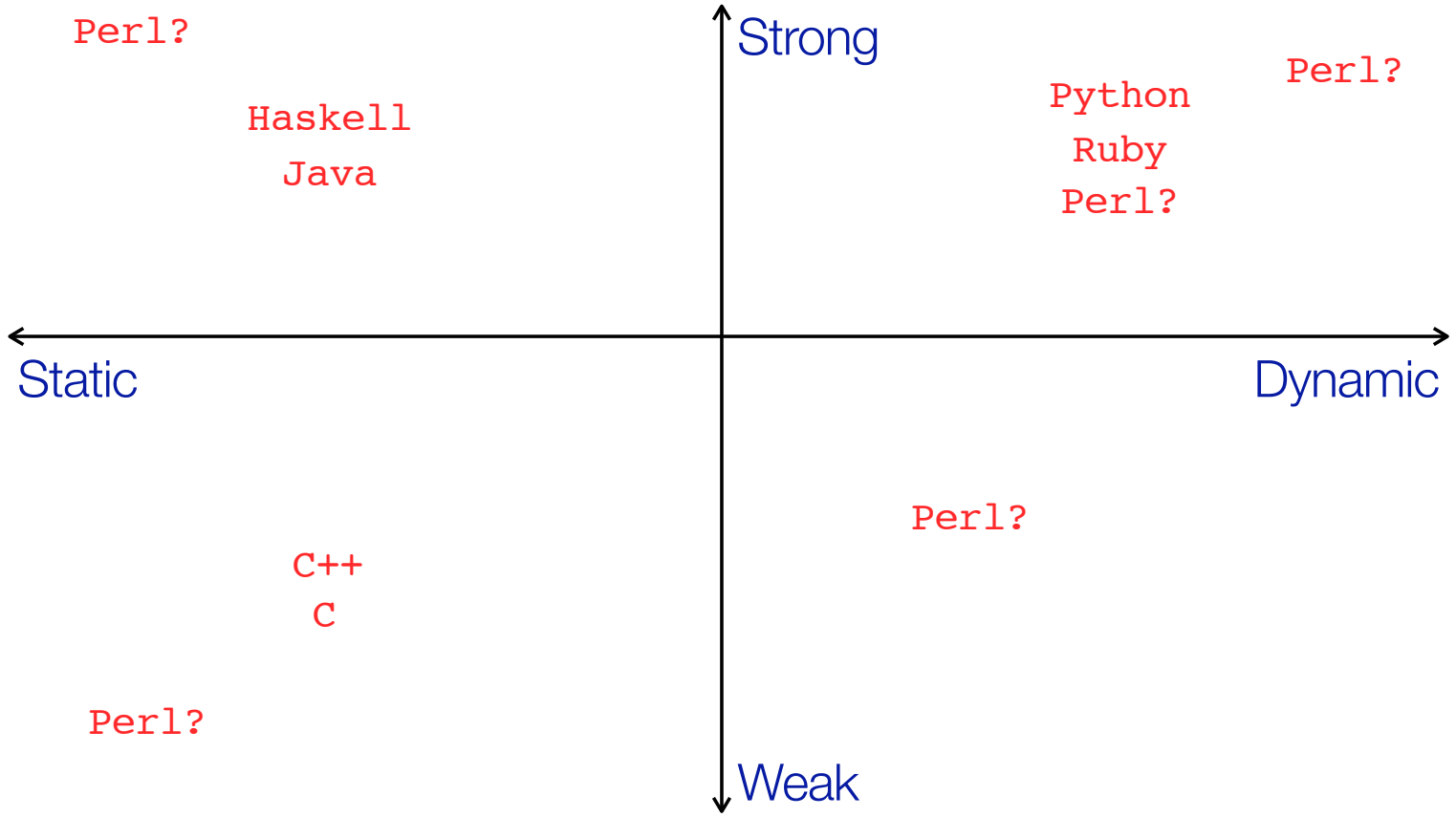
Static typing:

- Very (very very) often confused with strong typing
- Type constraints are checked at *compile-time*

Dynamic typing:

- Type constraints are checked at *run-time*
- Tag every value with its type
- Language's RTS (run-time system) checks tags are valid

Type Systems



Java: Welcome to the 60s

C, C++, Java uses static typing

- C, Java's static typing sucks (C++ discussed soon)
- Java forces you to do this:

```
Integer i = new Integer (69);
Vector v = new Vector ();
v.add(i); /* insert i into the vector named v */
…
Integer j = (Integer) v.get(0); /* get back i */
System.out.println (j.intValue() + 1);
```

- Java *forces* you to downcast the retrieved object to do anything useful with it
- C is even worse: cast is unsafe (weakly typed), ala callback example earlier

More Than a Syntax Problem

You can get run-time errors:

- C: segfault, or even worse, no segfault :-)
- Java: ClassCastException

Dynamic type checking solution:

- Python/Ruby use ‘duck typing’ (quacks like a duck, walks like a duck ⇒ is a duck)
- Pros: less syntax clutter
- Cons: still get run-time errors

Dynamic typing advocates say …

- Type systems restrict how I program
- Type system verification replaced by better testing
- Why the hell do I need these stupid type systems?

More Than a Syntax Problem

You can get run-time errors:

- **C: segfault, or even worse, no segfault :-)**
- **Java: ClassCastException**

Dynamic type checking solution:

- Python/Ruby use ‘duck typing’ (quacks like a duck, walks like a duck ⇒ is a duck)
- Pros: less syntax clutter
- Cons: still get run-time errors

Dynamic typing advocates say …

- Type systems restrict how I program
- Type system verification replaced by better testing
- Why the hell do I need these stupid type systems?

Parameterised Types

Static typing: use *parameterised types*, a.k.a. …

- Parameterised types in Haskell/ML

```
data List a = Cons a | Nil
```

- Templates in C++
- Generics in Java/C#

Benefit?

- Correctness *proved* at compile-time by type system
- Run-time error is *not possible*

Perl Poetry

```
#!/usr/bin/perl

APPEAL:

listen (please, please);

  open yourself, wide;
    join (you, me),
  connect (us,together),

tell me.

do something if distressed;

    @dawn, dance;
    @evening, sing;
    read (books,$poems,stories) until peaceful;
    study if able;

    write me if-you-please;

sort your feelings, reset goals,
seek (friends, family, anyone);

      do*not*die (like this)
      if sin abounds;

keys (hidden), open (locks, doors), tell secrets;
  do not I-beg-you, close them, yet.

          accept (yourself, changes),
          bind (grief, despair);

  require truth, goodness if-you-will, each
moment;

select (always), length(of-days)

# listen (a perl poem)
# Sharon Hopkins
# rev. June 19, 1995
```

(Updated for Perl 5.8.1-RC3 by me)

Type System Power

Static semantics & dynamic typing:

- Perl Poetry is syntactically valid
- … but it's *semantically* invalid

C/Java's static typing sucks, revisited:

- They suck because they have no *power*
- Cannot express anything beyond int, float, struct, union

Parameterised types add power to static semantics

- Core problem: type system not powerful enough
- Power = *expressiveness*

On OO & Static Typing

Inheritance & Static Typing Don't Mix:

- Derived class can add a new method '*foo*' to parent class
- If you have a value of the parent class's type, can you invoke *foo* method on that value? Maybe, maybe not …
- Safety dictates that you must choose 'maybe not'

The Rise of Dynamic Typing

- It's an object-oriented world
- Witness ease of coding in Python, Objective-C vs Java

What about C++ templates?

- C++ templates used for STL
- "*STL is not object oriented*" — Alex Stepanov

Type Systems Research

OO languages are stagnant w.r.t. type systems:

- Type system inherently limited by inheritance

Functional languages:

- Don't have inheritance (but do have OO-like features, e.g. type classes replace interfaces/dynamic binding/virtual methods—with zero run-time overhead)
- Where all the great type research has happened

All Praise Hindley-Mildner

Hindley-Mildner-Damas Type Inference:

- No need to annotate values with types
- Basis of modern functional languages (e.g. Haskell, ML)
- Type system becomes an *automated proof system*
- Functional languages go to *great* lengths to keep type inference
- Surprisingly often, if it's compiles, it's correct!
- Mark Jason-Dominus's "Strong Typing and Perl" talk: type checking found an infinite loop bug in a merge sort!

Types for Correctness

Dependent types:

- Parameterised types where the parameter can be a *value*
- Lists of specific length
- Matrices which are guaranteed to be square
- Balanced trees enforced by type system
- *Prove* that a program terminates

Types for Software Eng.

Advanced generics:

- Not to be confused with Java/C#'s generics
- Haskell/Perl has a *map* function, can be applied to lists:

```
map (* 2) [1..10]
```

- What about map'ping hash tables? Trees? *Any* type?
- Instantly add serialisation to all your new types
- Similar to what is possible with Python/Objective-C/Java/C# metaclasses/reflection, but *zero* run-time overhead

Types for Optimisation

Traditional optimisation

- e.g. Starkiller for Python

Linear types

- Compiler infers that a value is only used once
- Enables in-place updates of data:

```
char *get_scheme (char *url) {
  char *s = strstr (url, “://”);
  return strndup (url, s - url);
}
void main (int argc, char **argv) {
  char *url_scheme = get_scheme (argv[1]);
  printf (”scheme is %s”, url_scheme);
}
```

Proof-Carrying Code

Enables fast, safe execution of untrusted code

- Java can execute untrusted code in a ‘sandbox’ JVM
- Proof-carrying code promises to do the same, but again with *no* run-time overhead (well, not quite)

How?

- Attach types as *proof certificates* to object code
- Proof checker validates certificate (i.e. checks types)
- Generating proof certificate is slow, but checking it is fast

Benefits

- Guarantee a program will not delete your home directory
- Device drivers that are guaranteed not to panic the kernel
- Works even in the presence of a malicious compiler

Type Theory

$$\boxed{\Theta \Vdash \theta}$$

$$\frac{\theta \in \Theta}{\Theta \Vdash \theta}(mono) \qquad \frac{\Theta \Vdash \forall\alpha.\theta}{\Theta \Vdash [\tau/\alpha]\theta}(spec) \qquad \frac{\Theta \Vdash \pi \Rightarrow \phi \qquad \Theta \Vdash \pi}{\Theta \Vdash \phi}(mp)$$

$$\boxed{\Theta \vdash \sigma}$$

$$\frac{\Theta \Vdash D\,\tau \quad S \text{ is an associated type of D}}{\Theta \vdash S\,\tau} \qquad \frac{\Theta \vdash \sigma \qquad \alpha \notin \mathrm{Fv}(\Theta)}{\Theta \vdash \forall\alpha.\sigma} \qquad \frac{\Theta, \pi \vdash \rho}{\Theta \vdash \pi \Rightarrow \rho} \qquad \frac{\Theta \vdash \tau_1 \quad \Theta \vdash \tau_2}{\Theta \vdash \tau_1\ \tau_2} \qquad \frac{}{\Theta \vdash \alpha} \qquad \frac{}{\Theta \vdash T}$$

$$\boxed{\Theta \mid \Gamma \vdash e : \sigma}$$

$$\frac{(v : \sigma) \in \Gamma}{\Theta \mid \Gamma \vdash v : \sigma}(var) \qquad \frac{\Theta \mid \Gamma \vdash e_1 : \sigma_1 \qquad \Theta \mid \Gamma[x : \sigma_1] \vdash e_2 : \sigma_2}{\Theta \mid \Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \sigma_2}(let)$$

$$\frac{\Theta \mid \Gamma[x : \tau_1] \vdash e_2 : \tau_2 \qquad \Theta \vdash \tau_1}{\Theta \mid \Gamma \vdash \lambda x.e_2 : \tau_1 \to \tau_2}(\to I) \qquad \frac{\Theta \mid \Gamma \vdash e_1 : \tau_2 \to \tau_1 \qquad \Theta \mid \Gamma \vdash e_2 : \tau_2}{\Theta \mid \Gamma \vdash e_1\ e_2 : \tau_1}(\to E)$$

$$\frac{\Theta, \pi \mid \Gamma \vdash e : \rho}{\Theta \mid \Gamma \vdash e : \pi \Rightarrow \rho}(\Rightarrow I) \qquad \frac{\Theta \mid \Gamma \vdash e : \pi \Rightarrow \rho \qquad \Theta \Vdash \pi}{\Theta \mid \Gamma \vdash e : \rho}(\Rightarrow E)$$

$$\frac{\Theta \mid \Gamma \vdash e : \sigma \qquad \alpha \notin \mathrm{Fv}(\Theta) \cup \mathrm{Fv}(\Gamma)}{\Theta \mid \Gamma \vdash e : \forall\alpha.\sigma}(\forall I) \qquad \frac{\Theta \mid \Gamma \vdash e : \forall\alpha.\sigma \qquad \Theta \vdash \tau}{\Theta \mid \Gamma \vdash e : [\tau/\alpha]\sigma}(\forall E) \qquad \frac{\Theta \mid \Gamma \vdash e : \sigma}{\Theta \mid \Gamma \vdash (e :: \sigma) : \sigma}(sig)$$

On Objective-C

Get Mac OS X

- … *not* for the pretty GUI!
- To taste Objective-C and Cocoa programming
- GNUstep is good, but not as polished
- Read Apple's "The Objective-C Programming Language"

Small example:

- Use xine-lib media engine to create a media player
- GNOME's Totem: 27931 lines of C
- xine-ui (the main xine player): 96791 lines of C
- Cocoaxine on Mac OS X: **1436** lines of Objective-C
- apfelXine (Mac xine front-end): 2653 lines of Objective-C
- Build a media player/web browser in < 5 minutes!

Broaden Your Horizons

There's lots of fantastic technology out there!

- The Pragmatic Programmer recommends learning a new programming language each year

Language recommendations

- C: simplicity
- Objective-C: C + simple object system + dynamic typing
- Perl: integrated regular expressions
- Python: Objective-C + no memory management

Broaden Your Horizons

More language recommendations:

- C++: everything is possible, at the cost of complexity
- O'Caml: introduction to functional programming
- Haskell: introduction to type wizardry, correctness
- Nemerle: OO + functional programming (type inference!)
- Scheme/LISP: **macros**, subsume all these features and integrate them well into the language

Message of the talk:

- Don't be blinded by your own programming language community & advocacy

Links

Some opinions and rants …

- What I think about language shootouts/benchmarks:

  `http://xrl.us/cada`

- On object-orientation in non-object-oriented languages:

  `http://xrl.us/cadb`

- Mark Jason-Dominus on language advocacy:

  `http://www.perl.com/pub/a/2000/12/advocacy.html`

References

- Steve Blackburn's homepage (re garbage collection):

  `http://cs.anu.edu.au/~Steve.Blackburn/`

- "Programming Languages: Theory and Practice" — Bob Harper (type systems, type safety)

  `http://www-2.cs.cmu.edu/~rwh/plbook/`

Links

References …

- "In the Spirit of C" (article on B → C → C++/Java)

  `http://www.artima.com/cppsource/spiritofc.html`
- "Modern C++ Design" by Alexei Alexandrescu: design patterns enforced by C++ template wizardry

  `http://www.moderncppdesign.com/`
- "C++ Templates are Turing-Complete"

  `http://osl.iu.edu/~tveldhui/papers/2003/turing.pdf`
- Interview with Alex Stepanov (on STL and OO)

  `http://www.stlport.org/resources/StepanovUSA.html`
- Lambda the Ultimate—excellent programming languages blog (bias toward functional languages, though)

  `http://www.lambda-the-ultimate.com/`