# Dynamic Object-Oriented Languages

David Chisnall

# Dynamic Object-Oriented Languages

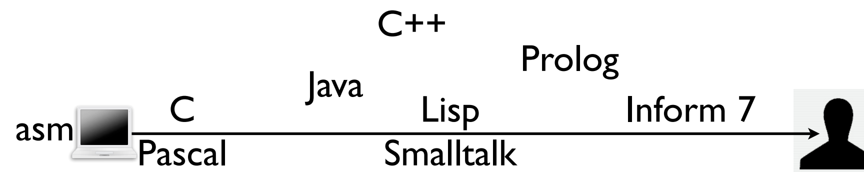
David Chisnall

# Talk Overview

- The Philosophy of Language Design.
- What are Dynamic Languages?
- What is Object Orientation?
- Ongoing work.

# Translation

- People think in one language.
- Computers 'think' in another.
- *Programming languages must be easy for humans to write and for computers to understand.*






Some languages (e.g. C++) are very hard to understand and technically impossible to compile.

# The Turing Tar Pit

- A Turing-complete language can implement any algorithm. Examples include:
  - A single-instruction with a subtract-and-branch-if-negative instruction.
  - PostScript, Excel, and XSLT
- Are these good general purpose languages?
- Turing completeness is *important* but not *interesting* when evaluating languages.



Turing completeness is important for a general-purpose language, but it's trivial to make even very small languages Turing complete. Other aspects are more important.

Most people who programmed in assembly languages built their own 'languages' from macros. Two possible (contradictory) good measures are:

1) How many users are able to use the language without building their own language on top.

2) How easy it is to build a new language on top.

Philosophical difference - dynamic languages favour definition 2.

'Syntactic Sugar' is not something to be disparaged - all languages are syntactic sugar.

## A Good Language

- Easy to write.
  - Easy to rewrite.
- Possible (ideally easy) to compile to fast code.



Hardly any code is written once and then thrown away. Code persists, often for decades beyond its original designers' intentions (see all the COBOL code still in use). A good language should make it easy for someone else to modify the code to add features later.
Being easy to compile fast (or interpret) is also important, since it allows much more interactive program development, which reduces bugs.
Static languages typically require a lot of static analysis at compile time, giving them a much longer write-compile-test cycle. Dynamic languages defer this to runtime so can be compiled or interpreted quickly.

## The Sapir–Whorf Hypothesis

- Postulates a causal relationship between natural language and thought patterns.
- More relevant to computer languages than natural ones:
  - Natural languages can easily steal parts from others if they aren't expressive enough.
  - "Notation as a tool of thought" - Kenneth E. Iverson, 1980 Turing Award Lecture.



Edward Sapir and Benjamin Whorf.
Also known as 'linguistic relativity hypothesis.'

# Software Libre has a certain je ne sais quoi.

What's the programming language equivalent of this construction?

Natural languages can steal words from each other. Programming languages can't.



Some languages can steal verbs from each other using foreign function interfaces, but writing parts of a program in one language and parts in another is often very hard.

# Dynamic Languages

# Informal Definition

- From Steve Yegge:
  - Ruby, Smalltalk, Lisp, Prolog.
  - ‘definition by example’

# Less Informal Definition

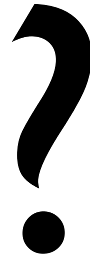
- Easy to write a metacircular evaluator i.e. it is simple to implement the language in itself (or a subset of itself).
- Communist syntax - no second-class citizens.
  - Turtles all the way down - everything is a list/function/object/etc. Homogeneous construction.



Static versus dynamic is a continuum, not a binary distinction. When we say ‘easy’ we mean ‘a language is more dynamic than another if it is easier’ rather than some arbitrary cut off between ‘easy’ and ‘not easy.’

Everything defined by the language can be replaced by the user.

## Formal Definition



12

Left as an exercise for the audience.
Some starting points - most programs are dynamic to a degree, since they operate on dynamic data. Three levels of dynamism: static, dynamic data, dynamic program.

# The Good

- Users can extend the language
  - No longer limited by the notions of the language designer.
- Example:
  - Foreach statement, map and fold all added to Smalltalk.



In a pure dynamic language, every single aspect of the semantics would be pluggable. This includes flow control (including concurrency), type systems, data and memory management strategies and so on.
No pure dynamic language exists.

# The Bad

- Hard to optimise
  - Most optimisations depend on compile-time invariants.



Heuristic optimisations such as speculative inlining are possible. More on this later.

*‘I made up the term "object-oriented", and I can tell you I did not have C++ in mind’*

- Alan Kay, OOPSLA 1997 keynote

# Objects

- Simple computers communicating by message passing.
- Local state exposed only via messaging.
- Ingalls' definition requires object types to be defined only by their interface.

# Smalltalk

- First pure OO language.
- Everything is an object.
- All interactions with objects are via message passing.

# Basic Syntax

- Unary message:
  - `a message.`
- Message with an argument:
  - `a messageWith:anArgument.`
- A message with two arguments:
  - `a messageWith:anArgument and:another.`

# Flow Control

- Blocks:
  - `[ :x :y | ...]`
- Equivalent to Lambda expression:
  - $\lambda xy. \ldots$
- Blocks are objects - respond to messages for binding and executing.

# If Statements

- Booleans are objects.
  - True and False are subclasses of Boolean.
  - Both respond to ifTrue: message
- A block is the argument.
  - True sends the argument an execute message.

# While Loops

```
whileTrue: aBlock
|a|
a := [self value].
a ifTrue:[aBlock value].
a ifTrue:[self whileTrue:aBlock].
^Nil.
```

# How Does It Work?

- Hardware has no concept of objects and messages.
- Message handlers (‘methods’) are functions with an implicit ‘self’ argument.

# Method Lookup

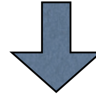
object doSomethingWith: anObject and: anotherObject

- Message sends have three components:
  - The receiver
  - The selector
  - The arguments

# Method Lookup

lookup(object, selector) → f(object,...)

object doSomethingWith: anObject and: anotherObject



method = lookup(object, #doSomethingWith:and:);

method(object, anObject, anotherObject);



Define runtime.

# The Lookup Function

- Static languages:
  - Lookup function evaluated at compile time.
- Dynamic languages:
  - Lookup function computed at runtime.
  - Implemented in a *virtual machine* or a *runtime library* (‘runtime’ for short)

# Delegation

- Fundamental part of most OO languages.
- An object specifies one or more other objects to provide message handlers if it does not provide them itself.
- Classes are a special case of delegates.

# Variations: Classes

- Lots of objects share the same selector to method mapping.
- Define a ‘class’ for each object.
- Make lookup depend on the class, rather than the object.

# Variation: Access Control

- Some methods should only be usable by some objects.
- Modify the lookup function to take the sender as an input.

# Variations: Extra Parameters

- Methods take self as a hidden parameter.
- Languages can also require other things.
  - Objective-C passes the selector (used for forwarding).



If a class doesn't implement a method, an ObjC runtime returns a generic function. This looks at the selector hidden argument and wraps up the invocation in an object which is then passed to the forwardInvocation: method.

# Ongoing Work:<br>A Modern Runtime

To appear in Journal of Object Technology

# Motivation: Étoilé

- Aims to build a desktop and post-desktop environment.
- Heavy use of dynamic languages (Objective-C, Smalltalk, Io).
- Finding existing tools too limiting.

David Chisnall, Swansea University

Quentin Mathe, Paris

Nicolas Roard, Google (formerly UWS)

Jesse Ross, Sevnthsin

The Étoilé Core Team



When there are only a few of you, you need to be a lot more efficient than the competition. Being much more clever than them is one way, but that's really hard. Having better tools is a much easier (and more scalable) way.

Yen-Ju Chen is missing from the core team because he still hasn't sent us a photograph.

# Generalised Dispatch

- No classes - pure prototype-based.
  - Classes built on top.
  - Any object can have methods added to it.
- Generalised lookup function:
  - lookup(receiver, type, selector, sender) $\rightarrow$ {receiver’, type’, method, context}



Modifying the receiver allows zero-cost local forwarding.
Modifying the types allows languages with no type information to perform auto-boxing.
The context allows extra information to be passed (e.g. in JavaScript the function object as well as the object)

# Generic Enough?

- Objective-C implementation (part of LLVM) 80% done.
- Smalltalk implementation 60% done.
- Implementations in progress by others:
  - Io - Quentin Mathe
  - JavaScript, ObjLisp - Alex Botero-Lowry (Reed College)

# Better Performance?

- Two sources of overhead:
  - Lookup function call.
  - Method function call.

## Avoiding Overhead 1

- Avoid the method function overhead with inlining
  - Requires knowing (at compile time) of what the function is.
  - Guessing works surprisingly well.
  - Type feedback works better.



Guessing comes from noticing that most methods are only implemented by a few objects.
Type feedback comes from Self – runs the code in an interpreter a few times to see what the types are.  Trace caching, from some modern JavaScript implementations is even better – compiles optimised versions for each type and branching at the start.

# Avoiding Overhead 2

- Avoid lookup overhead by caching the result of the lookup.
- Cache multiple object to method mappings at each call site (*Polymorphic Inline Caching*)
- Done in Self over a decade ago.
- Not done in compiled languages (until now)



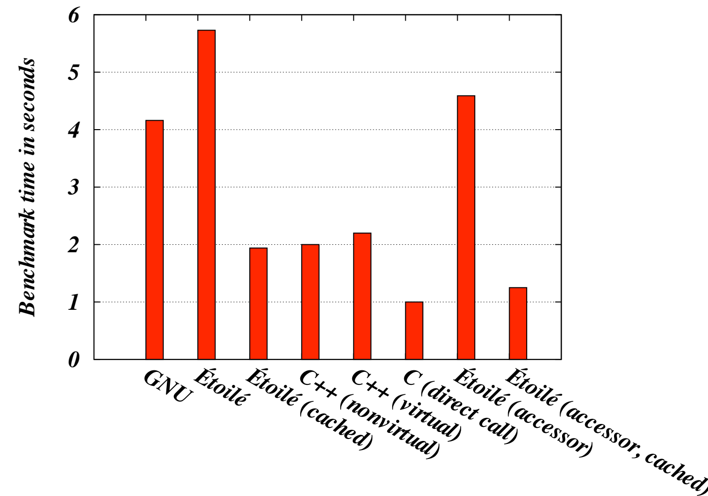
Caching lookup is hard to do safely in compiled lookups, because of the need to invalidate the cache when dispatch changes.

## PIC in Objective-C

- objc_msg_loopup() function returns function pointers.
- Caching these is easy.
- Knowing when the cache is invalid is harder.
- Solution: store a version with each pointer.



When you do a lookup, you cache a pointer to a structure containing the function pointer and the version, and the version. When you add a method, you first do a lookup, then increment the version. When you make a call, you compare the direct and indirect caches of the version, and if they match then you skip the lookup.





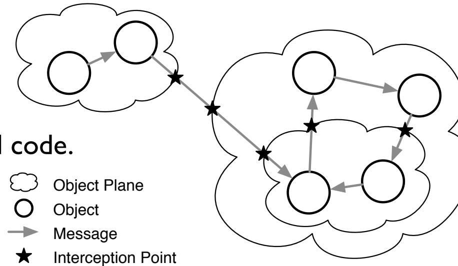

Figures are slightly out of date – recent optimisations gave about a 25% speed boost to the Étoilé runtime (it turns out branch instructions are really expensive).

# Even More Dynamic Dispatch

- Objects for runtime semantic groupings (e.g. a document, a window, and so on).
- Why not use this for message dispatch?

# Object Planes

- Messages can be intercepted by Plane objects when they cross a plane boundary.
- Uses:
  - Running untrusted code.
  - Language bridges
  - Concurrency






Processes are a special case of this general abstraction.