Open, extensible dynamic programming systems

or just how deep is the ‘dynamic’ rabbit hole?

Ian Piumarta
Viewpoints Research Institute
ian@squeakland.org
dynamic

dynamic?

• data?
  – extending objects and definitions
  – modifying the type system
    * modifying the dynamic type system

• behaviour?
  – extending the program during execution
    * modifying the dynamic effects of execution

• something that subsumes/unifies both, obviates one?

‘dynamic languages’ provide direct tools to do this
language

language?

• syntax
  – valid sentences of the language

• semantics
  – finding a meaning for those sentences

• implementation
  – translation of semantics into ‘more primitive’ equivalents

• runtime
  – the machine and its intrinsic mechanisms

• pragmatics
  – interaction with libraries, users, and everything

when?

• compile time? run time?

• staged? continuous?
dynamic languages

dynamic language?

• syntax
  – redefining ‘valid sentence’

• semantics
  – redefining or extending the range of meaning

• implementation
  – creating meaning from ‘primitive’ execution mechanisms
  – ability to directly modify machine code, including ...

• runtime
  – creating new ‘primitive’ execution mechanisms

• pragmatics
  – discovering, attaching to, using external resources

at any time during development, compilation, deployment, execution

The assertion that ‘a language is dynamic’ is more an assertion about the ease of use of dynamic features than it is a clear statement of the capabilities of the language. [WP]
techniques and tools

introspection

- from determining the type of a polymorphic value
- to full analysis of the system’s code as data

1st-class dynamic functions

- construction of new behaviour at any time
- immediate or delayed execution

active introspection (intercession)

- full access to the compiler
  - compiler implementation is 1st-class dynamic functions
- full access to the runtime
  - runtime mechanisms are 1st-class dynamic functions
- (re)define language elements
  - new syntax, grammar, semantics, optimisations
questions for this talk

how much of the language/system can we make dynamic?

how dynamic can we make it?

how general can we make it?

how simple can we make it?
less is more

It seems that perfection might be attained not when there is nothing left to add but rather when there is nothing left to take away.


extreme late binding

- less mechanism $\Rightarrow$ fewer assumptions to early bind
- fewer assumptions $\Rightarrow$ easier to late-bind everything
- eliminate early-bound assumptions $\Rightarrow$ generality
self-hosting system

- completeness
- understandability
- independence
- portability
objects

minimal object:
  • encapsulates behaviour
  • (subsumes state)

no assumptions about object contents
  • decouple implementation from representation
  • representation arbitrary
  • behaviour replaceable (and shareable)
  • implementation of behaviour replaceable (and shareable)
the minimal object

oop \rightarrow \textit{vtable}

\textit{increasing memory addresses} \downarrow

\approx \ ? \approx

send(message, object, ...) :=
method := object[-1].lookup(message)
method(object, ...)

vtable protocol

vtable.lookup(aSelector)
vtable.methodAtPut(aSelector, aMethodImplementation)
vtable.intern(aString)
vtable.allocate(objectSize)
everything is an object
methods as closures

- closure implementation can be shared
- closure data can be shared
- closure can reimplement ‘apply’
mixed-mode execution

• one ABI, multiple execution mechanisms
• transparent to sender
• profile-based optimisations (JIT), cross-paradigm calls, etc...

```c
interp(byteCodeArray *)
{
    ...
}
```

```
byte-compiled method 1
```

```
byte-compiled method 2
```
methods as value holders

- one getter one setter method, shared by all
- getter closure holds value
- setter closure assigns value of getter closure
- slot lookup as fast as method cache
revised vtable methods

after exposing the entire implementation:

```
selector    intern: nameString
closure     withMethod: aMethod data: anything
vtable      methodAt: aSelector put: aClosure
vtable      lookup: aSelector
vtable      allocate: objectSize
```

with intrinsic delegation mechanism (unnecessary but simplifies bootstrapping):

```
object      vtable     → self[-1]
vtable      delegate   → new vtable delegating to self
```

thus:

```
myPrototype := object vtable delegate allocate: 0

myPrototype new := [
   ↑self vtable allocate: self objectSize
]
```
static implementation of a dynamic universe

independent axes
• compilation (static/offline vs. dynamic/incremental)
• execution (static/early-bound vs. dynamic/late-bound)

```
rm -rf objComp.c
cc objComp.src
objComp.c
cc anything.src
object compiler
anything.exe
⇒ hello world
```
the object compiler

demo

• self-hosting static compiler for an extreme late-bound execution model

• dynamic environment creation
  
  vtable methodAt: aSelector put: aClosure

• dynamic execution mechanism
  
  vtable lookup: aSelector

⇒ reflect

object compiler (in itself): 2200 LOC

does it scale?

⇒ sqvm
objects are; methods do

consider Smalltalk

- represent ‘does’ with some ‘is’ (CompiledMethod)
- wave magic wand (apply VM to image)
- representation of ‘does’ indirectly moves messages around between the ‘is’

- methods have no dynamic effect without a VM
- the VM is not an object
- the actions of methods cannot be described purely in object terms
  - bind, apply, sequence

method objects imply how objects might be animated; the animation itself comes from ‘outside’

we need to bring it ‘inside’
objects are form

form

\[ O \times M \rightarrow I \]

\[ ( I = \{ O' \times M' \}^* ) \]
form needs function

form

\[ O \times M \rightarrow I \]

\[ ( I = \{ O' \times M' \}^* ) \]

function

\[ \lambda \\
01001011 \\
01100101 \\
11001000 \]
form describes function

form

\[ O \times M \to I \]

( \[ I = \{ O' \times M' \}^* \] )

function

\[ \lambda \]

01001011
01100101
11001000
functions transform form into function

form

\[ O \times M \rightarrow I \]

\[ ( I = \{ O' \times M' \}^* ) \]

function

\[ \lambda \]

01001011
01100101
11001000

© 2006 Ian Piumarta. Some rights reserved. For license terms see http://creativecommons.org/licenses/by-sa/2.5/
form describes function implements form

form

\[ O \times M \rightarrow I \]

\[ (I = \{O' \times M'\}^*) \]

expr -> IR -> gen

23

© 2006 Ian Piumarta. Some rights reserved. For license terms see http://creativecommons.org/licenses/by-sa/2.5/
functions

representation:s-expressions
• simple
• structured (eqv.AST)
• reasoning

domain:primitive operations and values
• closest to metal
• ‘animation’ of system (bind, apply, sequence) occurs at machine instruction level
• impedance with OS, libraries:C ABI

cf., Pre-Scheme (Richard Kelsey, 1997)
  LISP-70 (Larry Tesler et al., 1973)
semantics

evaluation

• atoms (42, "hello world") compile to primitive literal
• sequences evaluate elements and apply first to rest; unless ...
• sequence head defined as syntax ⇒ delegate compilation

(expression
  (... (myNodeType ...) ...)
(syntax myNodeType
  (lambda (node compiler)
    ;; rewrite and/or codegen and/or side-effect
    ...)))

• single mechanism for ‘intrinsic’ and ‘extended’ syntax
• intrinsic:quote syntax define set and or if while let lambda return send
  ⇒ operator ++
• syntactic sugar for send ⇒ seamless connection to representation
  ⇒ message send, definition
implementation

(printf "\%d\n" (+ 3 4))

| vpu entry |
entry := Label new.
(vpu := VPU new)
   define: entry;
   enter;
   ldInt: 3;
   ldInt: 4;
   add;
   ldPtr: '\%d\n';
   call: 2 target: (Label dlsym: 'printf');
   ret;
   compile.
↑ entry address
implementing semantics

(syntax while
  (lambda (node compiler)
    (let ((vpu [compiler vpu]))
      (or [[node size] == '3] (syntax-error node))
      [vpu begin: '2]
      [vpu br: '1]
      [vpu define: '0]
      [[node third] compile: compiler] ; body
      [vpu drop]
      [vpu define: '1]
      [[node second] compile: compiler] ; condition
      [vpu bt: '0]
      [vpu end: '2]
      [vpu ldInt: '0])
    0)))
complexity of implementation

transform s-expressions to VPU instructions

\texttt{compile.src}: 620 LOC (including intrinsics)

VPU implementation

\texttt{VPU.src}: 420 LOC
+ 176 LOC (ppc)
+ 118 LOC (i386)
\hline
714

© 2006 Ian Piumarta. Some rights reserved. For license terms see http://creativecommons.org/licenses/by-sa/2.5/
syntax and grammar

advanced recursive-descent parsing techniques (Birman, 1970)
• easy to write, read, understand
• more general and powerful than ‘traditional’ table-driver parsers
• just as amenable to analytic techniques

need to go back even earlier: Meta-II (Val Schorre, 1962)
• self-describing, self-implementing, self-bootstrapping dynamic syntax
• memoisation
• analytic treatments (predictive parsing)

synergy:
• T\(_E\)X (Donald Knuth, 1981)

demo:
• active parsing
• dynamic postfix operator syntax in a prefix language
everything is self-describing structure

downwards, sideways

& upwards

- lexical, syntactic, semantic, IR analysis as pattern-directed transformation
scorecard (so far)

- extend the program’s code during execution
- extend objects and definitions during execution
- a program analyzing its own structure, code, types or data
- executable data structures
- offline and online compilation
- VM, just-in-time compilation
- ability to directly modify machine code
- generating new objects from a runtime definition
- runtime alteration of object or type system
- changing the inheritance or type tree
- closures, continuations, introspection
- new language constructs, optimisations, grammar

<table>
<thead>
<tr>
<th>object comp</th>
<th>s-expr comp</th>
<th>VPU + 2 arch</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>2199</td>
<td>620</td>
<td>714</td>
<td>3533</td>
</tr>
</tbody>
</table>

© 2006 Ian Piumarta. Some rights reserved. For license terms see http://creativecommons.org/licenses/by-sa/2.5/
topics not discussed #1

implicit types

• tagged immediates
• NULL pointer as nil object

pragmatics

• primitive function __dlsym, primitive constant __RTLD_DEFAULT
• platform header parsing → dynamic interface generation

lazy coercions

• impedance mismatch between object and primitive type
• compiler-introduced message send to covert value
• ‘primitive methods’ become unnecessary
  – all ‘primitive’ behaviour subsumed by dynamic interfaces and coercions

selector-directed compilation
topics not discussed #2

reduction of compilation process to smallest number of ‘fundamental’ algorithms

- generic inference engine: parsing, compilation, codegen

```plaintext
reduce(tree, startSymbol) =
  foreach rule in startSets[startSymbol]
    if match(tree, rule.pattern)
      invoke rule.action
      return startSymbol

match(tree, pattern) =
  if (isSymbol(pattern)) return reduce(tree, pattern)
  if (tree.first ~= pattern.first) return false
  return foreach treeElement, patternElement in tree.tail, pattern.tail
    match(treeElement, patternElement)

ri4 :: (indiri4 (addp ri4 li2)) => $0 := $1 [ lwz r$1, $2(r$1) ]
ri4 :: (addp ri4 li2) => $0 := $1 [ addi r$1, $2 ]
ri4 :: (addi4 ri4 ri4) => $0 := $1 [ add r$1, r$1, r$2 ]
ri4 :: (cvtu4 ri4) => $0 := $1
ri4 :: (indiri4 (vregp)) => $0 := $1.1
li2 :: (cnstu4) ?? -32768 <= $1 <= 32767 => $0 := $1

(addi4
  (indiri4 (vregp 1)) => lwz r3, 12(r5)
  (indiri4 (addp (indirp (vregp 2))) => addi r3, r3, r4
  (cnstu4 12)))
```
putting it all together: parser

Stmt ::= 
    "{" Stmt*:s "}" => `(begin ,@s) 
    | "var" Binding:first (""," Binding)*:rest ";" => `(begin ,first ,@rest) 
    | "if" "(" Expr:c ")" Stmt:t ("else" Stmt | 
        Empty => '0):f => `(if (js-bool ,c) ,t ,f) 
    | "while" "(" Expr:c ")" Stmt:s => `(while (js-bool ,c) ,s) 
    | "do" Stmt:stmt "while" "(" Expr:cond ")" ";" => `(while (begin ,stmt ,cond)) 
    | "for" "(" ("var" Binding | Expr):init ";" 
        Expr:cond ";" Expr:upd ")" Stmt:s => `(begin ,init 
            (while ,cond 
                (begin ,s ,upd))) 
    | "break" ";" => '(break) 
    | "continue" ";" => '(continue) 
    | "return" (Expr:e => `(#return ,e) | 
        Empty => '(#return)):r ";" 
        Expr:e ";" => r 
        => e
(define js-set
  (lambda (lhs val)
    (match lhs
      ((js-get (js-get :c :n) :p)  '[(js-get ,c ,n) bind: ,p to: ,val])
      ((js-get (js-arr-get :a :i) :p) '[(js-arr-get ,a ,i) bind: ,p to: ,val])
      ((js-get :c :n) '[,]c set: ,n to: ,val])
      ((js-arr-get :a :i) '(js-arr-set ,a ,i ,val))
      (:otherwise  (error "%o is not assignable" lhs))))

putting it all together: semantics

JavaScript semantics: 100 LOC
function Array() {
    var l = arguments.length == 1 ? arguments[0] : 0;
    this.__array__ = #Array new: (js-get _ctxt 'l'));
    this.length = l;
    for (var idx = 0; idx < l; idx++)
        this[idx] = null;
}

JavaScript library: 102 LOC
(minimal Object, String, Date, Number, etc...)

© 2006 Ian Piumarta. Some rights reserved. For license terms see http://creativecommons.org/licenses/by-sa/2.5/
putting it all together: JavaScript

- semantics: 100
- parser: 86
- library: 102 (minimal Object, String, Date, Number, etc.)
- graphics: 136

⇒ js repl
⇒ cairo-app.js

just over 400 LOC
with no serious attempt at optimisation, runs a little faster than FireFox
(and a lot faster than WebKit aka Safari)
objects

- five ‘essential’ methods
- one semantic primitive (dynamic bind in the method cache)
- $\approx 2200$ LOC for self-hosting object compiler (Smalltalk-like syntax)
- infinitely extensible/reusable object framework
functions

- tightly integrated with object model $\Rightarrow$ dynamic objects
- compatible with platform ABI
- $\approx 1300$ LOC for self-hosting dynamic function $\rightarrow$ native code compiler
- infinitely extensible/reusable semantic framework
conclusion #3

take away:

• ‘dynamic’ can apply to everything (data, code, types, ...)
• ‘language’ can mean all of it (syntax, semantics, implementation, pragmatics, ...)
• it can be made very, very simple
• it can be made very, very general
• it can free you from a multitude of pedantries
• it is a lot of fun

attractive implementation vehicle

• active protocols, extensible/configurable systems
• programming/scripting languages
conclusion #4

ideal testbed for new language and systems ideas

- if you aren’t using dynamic languages this way, you should be

ideal for teaching language and systems principles

- if you aren’t teaching languages this dynamic, you should be

go home and innovate!

- built your own and share it with us
- or use ours: release in a month or two
Form follows function — that has been misunderstood. Form and function should be one, joined in a spiritual union.

Frank Lloyd Wright, 1908

It is the grand object of all theory to make these irreducible elements as simple and as few in number as possible, without having to renounce the adequate representation of any empirical content whatever.

Albert Einstein, Mein Weltbild, 1934

On the contrary, most of our systems are much more complicated than can be considered healthy, and are too messy and chaotic to be used in comfort and confidence.

Edsger W. Dijkstra, CACM 44(3), 2001