Why dynamic languages can and should address the world

Stephen Kell
King’s College London
stephen@humprog.org

October 2023
Thank you, apologies...
“Software seems ‘large’ and ‘complicated’ for what it does.”

Alan Kay
“If we succeed in making an Intergalactic Network, then our main problem will be learning to communicate with Aliens.”

J.C.R. Licklider
“Programming is 95% glue code.”

attribution unknown (anyone?)

... and it’s only going to get worse! (If...)
```python
png_file = fopen("dices.png", "rb")
png_reader = png_create_read_struct(PNG_LIBPNG_VER_STRING, None, None, None)

png_info = png_create_info_struct(png_reader)
png_init_io(png_reader, png_file)
png_read_info(png_reader, png_info)

width = png_get_image_width(png_reader, png_info)
height = png_get_image_height(png_reader, png_info)
row_bytes = png_get_rowbytes(png_reader, png_info)

color_type = ord(png_get_color_type(png_reader, png_info))
bit_depth = ord(png_get_bit_depth(png_reader, png_info))

if color_type != PNG_COLOR_TYPE_RGBA and bit_depth != 8:
    raise RuntimeError("png/uni2423file/uni2423do/uni2423not/uni2423have/uni2423the/uni2423right/uni2423image/uni2423format")
```

Why I like dynamic languages (1)

The dynamic style is

- succinct
- flexible
- usable

Dynamic languages should be *net removers of glue code*. 
About succinctness and flexibility . . .

In a hypothetical dynamic language, if I write:

```
x.y.foo()
```

I mean ‘invoke the operation foo associated with the data item known as y by its association with data item x’

- high-level! working with the logical structure of data
- can hide complexity, e.g. layout details
‘Invoke the operation \texttt{foo} associated with the data item known as \texttt{y} by its association with data item \texttt{x}’

Very flexible about what the logical structures \textit{are}!

- not only abstracting away \textit{the} concretion’s details. . .
- can deal with heterogeneity
- i.e. can hide \textit{diversity}
- while retaining \textit{interoperability} (sense of Aldrich)
- ‘find commonality where it exists!’—the essence of late binding

Contrast with, say, any language grappling with Hindley–Milner-esque type-soundess. . .
Why I like dynamic languages (2)

The world is dynamic!

- respects few invariants
- prone to data change, schema change, . . .
- large, yet explorable

Most useful code involves interacting with the (material) world.

- calling it ‘glue’ undervalues it

Surely dynamic languages can do this in a succinct, direct, usable way?
Why I dislike dynamic languages

I’m trapped in a box! A pleasant dynamic box, but. . .
Expectation: ‘you’ll like it so much in here. . .’
Why I dislike dynamic languages

Standard libraries only help a little. . .
‘Inverting the box’ is a latent, recurring goal of mine... 

Language should be the window to the outside!
png_file = fopen("dices.png", "rb")
png_reader = png_create_read_struct(PNG_LIBPNG_VER_STRING, None, None, None)
png_info = png_create_info_struct(png_reader)
png_init_io(png_reader, png_file)
png_read_info(png_reader, png_info)

width = png_get_image_width(png_reader, png_info)
height = png_get_image_height(png_reader, png_info)
row_bytes = png_get_rowbytes(png_reader, png_info)

color_type = ord(png_get_color_type(png_reader, png_info))
bit_depth = ord(png_get_bit_depth(png_reader, png_info))
if color_type != PNG_COLOR_TYPE_RGBA and bit_depth != 8:
But we didn’t have to write anything like…

```c
static PyObject *
spam_system(PyObject *self, PyObject *args)
{
    const char *command;
    if (!PyArg_ParseTuple(args, "s", &command))
        return NULL;
    int sts = system(command);
    return PyLong_FromLong(sts);
}
```
The main trick: make memory more self-describing...
The main trick: make memory more self-describing...
Provide ways to discover what’s there...

```
struct uniqtype;  /* type descriptor */
struct allocator; /* heap, stack, static, etc */
allocator * alloc_get_allocator (void *obj); /* which one? (at leaf) */
uniqtype * alloc_get_type (void *obj); /* what type? */
void * alloc_get_site (void *obj); /* where allocated? */
void * alloc_get_base (void *obj); /* base address? */
void * alloc_get_limit (void *obj); /* end address? */
Di_info alloc_dladdr (void *obj); /* dladdr-like */
```

// more calls go here...

This is (some of) the API of liballocs, my main project
We should even be able to near-eliminate stuff like this...

```python
import json
my_list = [];

with open('data.txt') as f:
    lines = f.readlines() # list containing lines of file
    columns = [] # To store column names

    i = 1
    for line in lines:
        line = line.strip() # remove leading/trailing white spaces
        if line:
            if i == 1:
                columns = [item.strip() for item in line.split(',')] # dictionary to store file data (each line)
                data = [item.strip() for item in line.split(',')] # list containing lines of file
                for index, elem in enumerate(data):
                    d[columns[index]] = data[index]
                my_list.append(d) # append dictionary to list
```
Taking these ideas further

Building an FFIless Python seems cool.

But is there a deeper take on these ideas?

Relatively few programming systems are ‘outward-facing’ in this way...

It seems necessary also to take some iconoclastic steps

- prioritise human efficiency over machine efficiency
- prioritise integration over computation

I’ve thought of a few programming systems that do some of these...
| LOGIN: KEN   |
| PASSWORD: KEN  |
| LS -L SYSTEM   |
| 00004 DRWR- 04 777 00050 |
| 00006 SRWR- 01 777 00000 1000000 |
| 00007 SRWR- 01 777 00000 KEYBOARD |
| 00010 SRWR- 01 777 00000 PPTIN |
| 00013 SRWR- 01 777 00000 FIT1001 |
| 00014 SRWR- 01 777 00000 DISPLAY |
| 00015 SRWR- 01 777 00000 |

photo: Living Computers: Museum + Labs
(https://www.youtube.com/watch?v=pvaPaWyiuLA)
Hold [Start] for reset
*** WozMon
00.2f

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>00 80 03 30 05 A1 D5 DA</td>
</tr>
<tr>
<td>0008</td>
<td>AA 8E 00 30 00 F3 0B FF</td>
</tr>
<tr>
<td>0010</td>
<td>FF FF 7F 04 04 1A B1 03</td>
</tr>
<tr>
<td>0018</td>
<td>18 00 B5 03 FE 47 02 4D</td>
</tr>
<tr>
<td>0020</td>
<td>B8 1C E1 04 20 0F 00 69</td>
</tr>
<tr>
<td>0028</td>
<td>1F 40 5B 20 00 08 0A 0A</td>
</tr>
<tr>
<td>600</td>
<td>de ad be af</td>
</tr>
<tr>
<td>0600</td>
<td>11</td>
</tr>
<tr>
<td>600.603</td>
<td></td>
</tr>
<tr>
<td>0600</td>
<td>DE AD BE AF</td>
</tr>
</tbody>
</table>
(gdb) print printf
$1 = \{\text{int (const char *, ...)}\} \ 0x7fffff7e344c0 <\_\_printf>
(gdb) set x = 42 + printf("Hello")
(gdb) print x
47
It’s not only dynamic languages that can be outward-facing. . .
We present a case study (Section 2) showing how a combination of numerous F# language features can be used for the development of modern web applications. This is not a toy demonstration, but an example of how F# is used in industry.

We discuss how type providers make it possible to access external information sources in web applications (Section 3.2) and integration with (untyped) programming environments such as the JavaScript ecosystem (Section 3.3).

We show how F# approaches the problem of compilation to JavaScript using a library called FunScript (Section 4.1), outlining important practical concerns such as interoperability (Section 4.2) and asynchronous execution (Section 4.2).

Throughout the paper, we discuss how the age of the web breaks the assumptions commonly taken for granted in typed functional programming. We revisit the notion of type safety in the context of the web (Section 5.1) and the notion of fixed language semantics (Section 5.2).

In the first part of the paper (Section 2), we present a case study of using F# for web development. The rest of the paper (Section 3 and 4) discusses the arising issues in more depth. The source code and running demo for the case study is available at: http://funscript.info/samples/worldbank

2 Case Study: Web-based data analytics

In this case study, we develop a web application shown in Figure 1, which lets the user compare university enrollment in a number of selected countries and regions around the world. The resulting application runs on the client-side (as JavaScript) and fetches data dynamically from the World Bank [1].

The application is an example of a web page that could be built in the context of data journalism [13]. As such, it is relatively simple, works with just a single data source and uses a concrete indicator and a hard-coded list of countries i.e. to illustrate a point made in an accompanying article.

2.1 Accessing World Bank data with type providers

To access the university enrollment information, we first obtain a list of countries using the World Bank type provider from the F# Data library [20]. The type provider exposes the individual countries as members of an object (the notation `Country Name` is used for identifiers with spaces):

```fsharp
type WorldBank = WorldBankData<Asynchronous = true>

let data = WorldBank.GetDataContext()
let countries =
    [ data.Countries.```European Union``
    data.Countries.```Czech Republic``
    data.Countries.```United Kingdom``
    data.Countries.```United States`` ]
```
// all valid identifiers...

person

name

var:person/name

var:person/{attribute}

file://tmp/button.png

http://www.example.com/button.png

file:{env:HOME}/rfcs/{rfcName}
The logical end point

‘Data in’ and ‘data out’ entail glue code because *data within* is different

The ‘objects’ that our ‘language’ works with are special!

Do they have to be?

// our earlier Python interpreter

class ParathonValue { /* 9 fields, 25 methods */ };

// in DwarfPython
typedef void ParathonValue;
Can we base our data model on a data description language?

ClassFile {
    uint32 magic = 0xCAFEBAEB;
    uint16 minor_version = 3;
    uint16 major_version = 45;
    uint16 cp_count;
    ConstantPoolInfo constant_pool[1..cp_count];

    bitmask uint16 ClassFlags {
        ACC_PUBLIC, ACC_FINAL, ACC_ABSTRACT, ACC_INTERFACE, ACC_SUPER
    } access_flags;

    ...

The idea redux:

- no such thing as ‘object layout’
- use our language to access any recognisable structure
- recovering efficient access patterns is for the JIT

example: Godmar Back’s Datascript from GPCE ’02, encoding the Java class file format
Can we apply the same ‘zero knowledge’ idea to operations?

What does it mean to operate on program state?

Can we bootstrap a language that can be ‘taught’ all compound operations?

 Raises the question of what a language ‘core’ should contain.

It should let us access the outside world, of course!

// before-state
@ syscall: (*(uint$8(*)[2])m.rip) == INT_0x80 {
  m.rax <-> number : word,
  m.rdx <-> arg1 : word,
  ...
}
// after-state
@
 syscall: (*(uint$8(*)[2])(m.rip - 2) == INT_0x80 {
  m.rax <-> retval : word
}
syscall_descrs = { [1] /* write */ { arg1 <-> fd : int$32,
Can we apply the same ‘zero knowledge’ idea to operations?
Seeing the difference: VM context

// original Parathon version --- "standard design"
ParathonValue* FunctionCall::evaluate(ParathonContext& c)
{ return call_function ( this->base_phrase->evaluate(c),
                        this->parameter_list->asArgs(c) ); }

// less obtrusive DwarfPython version
val FunctionCall::evaluate() // ← context is the process stack
{ return call_function ( this->base_phrase->evaluate(),
                        this->parameter_list->asArgs() ); }

Avoid reinventing structures at VM level!
Also lets you do things like... 

```c
struct ellipse {
    struct point {
        double x, y;
    } ctr;
    double maj;
    double min;
} my_ellipses[3];

double p_d = &p_e->ctr.x

struct ellipse {
    struct point {
        double x, y;
    } ctr;
    double maj;
    double min;
} my_ellipses[3];
```
... dynamic checking the correctness of things like this

```c
struct ellipse {
    struct point {
        double x, y;
    } ctr;
    double maj;
    double min;
} my_ellipses[3];
p_f = (ellipse*) p_d
```

```c
struct ellipse {
    struct point {
        double x, y;
    } ctr;
    double maj;
    double min;
} my_ellipses[3];
```
... while avoiding classic problems of C bounds checkers

```
struct ellipse {
    struct point {
        double x, y;
    } ctr;
    double maj;
    double min;
} my_ellipses[3];
```