Composing the uncomposable
Some work, work-in-progress and ideas.

Stephen Kell

stephen.kell@cs.ox.ac.uk
Software is expensive and *inflexible*.

Tools assume:

- ground-up
- perfect fit
- homogeneous
- never change, never replace

Reality: none of the above!
Outline

- The Cake language
- Interface styles
- DwarfPython
- Interface hiding
the Cake language
interface styles
DwarfPython
interface hiding
These programming tasks arise:

- as software evolves
- as new user requirements emerge
- as software ecosystem evolves
  - e.g. alternative components become available
Cake in one slide

Cake is a tool for tasks like these. It is:

- a rule-based language
- ... for describing adapters
- declarative
- black-box
- convenient
Common approaches

- **Edit or patch**
  - Diagram showing a red cross indicating an edit or patch operation on components A and A'.

- **Glue coding**
  - Diagram showing glue code inserted between A and B.

- **Abstraction layer**
  - Diagram showing an abstraction layer for A, possibly with a target for original A.
exists elf_reloc ("A.o") A; // assume that object files...
exists elf_reloc ("B.o") B; // ... contain debug info
derive elf_reloc ("whole.o") = link[A, B]
{
    /* your rules here! */
};
Interlude: linking
Simple adaptations

\[
\text{foo} (...) \leftrightarrow \text{bar} (...);
\]
foo (...) ←→ bar (...);

baz(a, b) ←→ baz(b, a);
foo (...) $\leftrightarrow$ bar (...);

baz(a, b) $\leftrightarrow$ baz(b, a);

xyz(a) $\leftrightarrow$ xyz(a, 42);
Simple adaptations

foo (...) $\leftrightarrow$ bar (...);

baz(a, b) $\leftrightarrow$ baz(b, a);

xyz(a) $\leftrightarrow$ xyz(a, 42);

values Br $\leftrightarrow$ Tr
Simple adaptations

The Cake compiler generates wrapper functions from rules.
More complex adaptations

Real interfaces correspond less simply:

- non-1-to-1 mappings
- context-sensitive
- data, not just code
Many-to-many mappings

values (dec: mpeg2_dec_s, info: mpeg2_info_s, seq: mpeg2_sequence_s, fbuf: mpeg2_fbuf_s)
Many-to-many mappings

values (dec: mpeg2_dec_s, info: mpeg2_info_s, 
seq: mpeg2_sequence_s, fbuf: mpeg2_fbuf_s) 

←→

( ctxt : AVCodecContext, vid_idx: int, frame: AVFrame, 
p: AVPacket, s: AVStream, codec: AVCodec)
values (dec: mpeg2_dec_s, info: mpeg2_info_s, seq: mpeg2_sequence_s, fbuf: mpeg2_fbuf_s)

\[
\begin{array}{c}
\text{ctxt} \in \text{AVCodecContext}, \\
\text{vid_idx} \in \text{int}, \\
\text{frame} \in \text{AVFrame}, \\
\text{p} \in \text{AVPacket}, \\
\text{s} \in \text{AVStream}, \\
\text{codec} \in \text{AVCodec}
\end{array}
\]

\{ seq.width \leftrightarrow ctxt.width ; \}
Many-to-many mappings

values (dec: mpeg2_dec_s, info: mpeg2_info_s,  
seq: mpeg2_sequence_s, fbuf: mpeg2_fbuf_s)  

\[
\leftrightarrow \\
( ctxt : AVCodecContext, vid_idx: int, frame: AVFrame,  
p: AVPacket, s: AVStream, codec: AVCodec) \\
\}

\[
\begin{align*}
\text{seq.width} & \leftrightarrow \text{ctxt.width} ; \\
\text{seq.display_width} & \leftrightarrow \text{frame.linesize}[0]; \\
\end{align*}
\]

/* ←- more rules go here... */
Context-sensitive mappings

Trace of start-up calls appropriate for the two libraries:

mpeg2_init() \mapsto \text{dec}
\text{fopen("...", "rb")} \mapsto \text{f}
mpeg2_info(dec) \mapsto \text{info}
mpeg2_parse(dec) \mapsto \text{STATE_BUFFER}
avcodec_init() \mapsto ()
av_register_all() \mapsto ()
av_open_file(...) \mapsto \text{avf}
av_find_stream_info(\text{avf}) \mapsto ()
avcodec_find_decoder(...) \mapsto \text{dc}
av_codec_open(...) \mapsto ()

Problem!

- one \text{info} call wants decoder object; other wants file.

Solution: context predication with name binding.

\begin{verbatim}
let dec = mpeg2_init(), ..., 
let f = fopen(fname, "rb"), ..., 
mpeg2_info(dec) \mapsto \{ /* now both f and dec are available */ \};
\end{verbatim}
Complex object structures

“Passing objects” often means passing object graphs.

The Cake runtime traverses object structures automatically.

```c
values list_node_t <-> ListNode
{
    data <-> item;
};
values point_t <-> XYPoint;
```

Incidentally, note the following:

- name-matching is Cake’s default policy
- can insert stub code for value transformation (not shown)
Implementation

Compiler:

- accepts Cake source file
- emits wrapper functions as C++ code
- consumes DWARF debugging information

Runtime:

- allocator instrumentation
- dynamic points-to analysis
- “split heap” management, association tracking, ...

Status: compiler still lagging language design, but WIP...
Compare Cake implementations with pre-existing adapters:

1. **p2k**: a filesystem adapter from the NetBSD OS
2. **ephy-webkit**: abstraction layer from Epiphany browser
3. **XCL** (subset of): compatibility layer for XCB X11 library

Summary outcome:

- less code (code *written*; various syntactic measures)
  - **p2k**: approx 70% reduction
  - **ephy-webkit**: approx 65% reduction
  - **XCL**: approx 30% reduction
- less scattering of concerns
int seek(struct puffs_usermount *pu, puffs_cookie_t opc, off_t oldoff, off_t newoff, struct puffs_cred *pcr) {
    kauth_cred_t cred; int rv;

    cred = cred_create(pcr);
    VLE(opc);

    rv = RUMP_VOP_SEEK(
        opc, oldoff, newoff, cred);
    VUL(opc);
    cred_destroy(cred);

    return rv;
}

int remove(struct puffs_usermount *pu, puffs_cookie_t opc, puffs_cookie_t targ, struct puffs_cn *pcn) {
    struct componentname *cn; int rv;

    cn = makecn(pcn);
    VLE(opc);
    rump_vp_incref(opc);
    VLE(targ);
    rump_vp_incref(targ);

    rv = RUMP_VOP_REMOVE(
        opc, targ, cn);
    AUL(opc);
    AUL(targ);
    freecn(cn, 0);

    return rv;
// rules concerning functions
p2k_node_seek(_, vn, oldoff, newoff, cred)
   → RUMP_VOP_SEEK(vn, oldoff, newoff, cred);

p2k_node_remove(_, vn as vnode_bump, tgtvn as vnode_bump, cn)
   → RUMP_VOP_REMOVE(vn, tgtvn, cn);

// rules concerning values
values puffs_cookie_t → ({VLE(that); that}) vnode;
values puffs_cookie_t ← ({VUL(that); that}) vnode;
values vnode_bump → ({VLE(that); rump_vp_incref(that); that}) vnode; // also bump refcount
values vnode_bump ← vnode; // unlock not required
values puffs_crd (cred_create(this)) → kauth_crd;
values puffs_crd ← (cred_destroy(this)) kauth_crd;
values puffs_cn (makecn(this)) → component_name;
values puffs_cn ← (freecn(this, 0)) component_name;

+ these rules contribute to other wrapper functions (28 total)
Related work

Similar tools with narrow domains or less expressiveness:
- Nimble (Purtilo, 1990), BCA (Keller, 1998)
- Jigsaw (Bracha, 1993), Knit (Reid, 2000)
- Swig (Beazley, 1996)
- C++ concept maps (Jarvi, 2007)
- Twinning (Nita, 2010)

Work focused on formalisation rather than implementation:
- Yellin & Strom, 1994; Bracciali, 2003
- subject-oriented composition (Ossher, 1995)

Clean-slate approaches to similar problems:
- Flexible Packaging (Deline, 2001)
Future work

So far, Cake is

- a simpler way of writing short modular adapters
- a convenient tool for binary composition
- a step towards more compositional development

Cake is a platform for lots of potential future work.

- styles (for abstracting heterogeneous object code)
- white-box complement
- improved bidirectionality
- semi-automatic generation of Cake rules
Things I didn’t have time to mention

More language features:

■ input versus output parameters
■ error discovery & handling
■ design for heterogeneity
■ stub language (algorithms, lambdas, …)
■ annotations, memory management adaptations, …

Repositories:

■ http://www.cl.cam.ac.uk/~srk31/cake/

Questions about Cake?
How Cake wins

- separate treatment of values from treatment of functions
- separate general from special cases
- name-matching
- black-box, binary, language-independent
- designed to accommodate heterogeneity
- make previously edit-requiring tasks black-boxable
- generates sequence recognition code automatically
- maintains object mappings (co-object relation) at runtime
- transitive treatment of object structures
- potential for bidirectional rules
Partially split heap

```c
void gtk_entry_set_text (e, t) {
  e->text = g_strdup(t);
  // ...
  return;
}
```

Library call directed into wrapper

Interposed code

Co-object

Switch accessible heap

LibGTK20 accessible heap

Switch binary

... gtk_entry_set_text (e, "..."); ...

Library function with alternate-rep "co-objects"

Return to client

Composing... – p.22/68
### Numbers

<table>
<thead>
<tr>
<th>p2k</th>
<th>C</th>
<th>adjusted</th>
<th>Cake</th>
<th>remaining C</th>
</tr>
</thead>
<tbody>
<tr>
<td>LoC (nb nc)</td>
<td>605</td>
<td>523</td>
<td>133</td>
<td>54</td>
</tr>
<tr>
<td>tokens</td>
<td>3469</td>
<td>3137</td>
<td>1131</td>
<td>347</td>
</tr>
<tr>
<td>semicolons</td>
<td>358</td>
<td>277</td>
<td>69</td>
<td>33</td>
</tr>
</tbody>
</table>

- wins: rule localisation (hugely), allocation

<table>
<thead>
<tr>
<th>ephy</th>
<th>C</th>
<th>adjusted</th>
<th>Cake</th>
<th>remaining C</th>
</tr>
</thead>
<tbody>
<tr>
<td>LoC (nb nc)</td>
<td>525</td>
<td>513</td>
<td>161</td>
<td>0</td>
</tr>
<tr>
<td>tokens</td>
<td>2529</td>
<td>2455</td>
<td>784</td>
<td>0</td>
</tr>
<tr>
<td>semicolons</td>
<td>175</td>
<td>163</td>
<td>70</td>
<td>0</td>
</tr>
</tbody>
</table>

- wins: rule localisation, many-to-many, graph exploration, pattern-matching
### XCL

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>adjusted</th>
<th>Cake</th>
<th>remaining C</th>
</tr>
</thead>
<tbody>
<tr>
<td>LoC (nb nc) tokens</td>
<td>380</td>
<td>315</td>
<td>189</td>
<td>42</td>
</tr>
<tr>
<td>tokens</td>
<td>2581</td>
<td>2328</td>
<td>1543</td>
<td>232</td>
</tr>
<tr>
<td>semicolons</td>
<td>187</td>
<td>148</td>
<td>107</td>
<td>19</td>
</tr>
</tbody>
</table>

- problems: abstraction gap, cross-rule commonality, more data types, more special cases, smaller subset of code (increasing returns?)
■ the Cake language
■ interface styles
■ DwarfPython
■ interface hiding
We have a lot of choices about how to write our software.

What if we didn’t choose the same $x$?

<table>
<thead>
<tr>
<th>$x$</th>
<th>I say …</th>
<th>You say …</th>
</tr>
</thead>
<tbody>
<tr>
<td>language</td>
<td>GLib</td>
<td>Java</td>
</tr>
<tr>
<td>libraries</td>
<td>return EIO;</td>
<td>throw new IOException();</td>
</tr>
<tr>
<td>conventions</td>
<td>while (it != end());</td>
<td>while (it.hasNext());</td>
</tr>
<tr>
<td>patterns</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Let’s call the whole thing off!
struct wc;
   // opaque to client

struct wc *wc_new(const char *fname);
   // returns NULL and sets errno on error

int wc_get_words(struct wc *obj);
int wc_get_characters(struct wc *obj);
int wc_get_lines(struct wc *obj);
int wc_get_all(struct wc *obj,
   int *words_out, int *chars_out,
   int *lines_out);

void wc_free(struct wc* obj);

Goal: link a client with its alternate style implementation
Simple example

```c
struct wc;
// opaque to client

struct wc *wc_new(const char *fname);
// returns NULL and sets errno on error

int wc_get_words(struct wc *obj);
int wc_get_characters(struct wc *obj);
int wc_get_lines(struct wc *obj);
int wc_get_all(struct wc *obj,
               int *words_out, int *chars_out,
               int *lines_out);

void wc_free(struct wc* obj);
```

```java
class WordCounter {
    /* fields not shown... */

    public WordCounter(String filename)
        throws IOException {
        /* ... */
    }

    public int getWords() { /* ... */
    }

    public int getCharacters() { /* ... */
    }

    public int getLines() { /* ... */
    }

    public Triple<Integer, Integer, Integer> getAll () { /* ... */
    }

    }
```

Goal: link a client with its alternate style implementation
What is a style?

■ It’s any set of interface conventions...
■ ... that recur across many components

Why do we care about styles?

■ By describing them explicitly in some way ...
■ ... we can link components that use different styles...
■ ... with less per-composition effort

How do we identify styles?

■ Empirically! By looking at existing code...
■ ... through a unifying lens.
A lot of code is (can be) compiled down to object code.

```
wc_new("README") = 0x9cd6180[struct wc]
```

```
wc_get_words(0x9cd6180[struct wc]) = 311
wc_get_characters(0x9cd6180[struct wc]) = 2275
wc_get_lines(0x9cd6180[struct wc]) = 59
wc_get_all(0x9cd6180[struct wc], 0xbfffeed00[stack],
            0xbffeecfc[stack], 0xbffeecf8[stack]) = 0
wc_free(0x9cd6180[struct wc]) = ()
```

- Traces (including data structures) are our unifying lens.
A lot of code is (can be) compiled down to object code.

```c
_Jv_InitClass(..., 0x6015e0[java::lang::Class], ...) = ...
_Jv_AllocObjectNoFinalizer(..., 0x6015e0, ...) = 0x9158d20
WordCounter::WordCounter(java::lang::String*)(
    0x9158d20[WordCounter], 0x9ae3dc8[java::lang::String])
    = ()
WordCounter::getWords()(0x9158d20[WordCounter]) = 311
WordCounter::getCharacters()(0x9158d20[WordCounter]) = 2275
WordCounter::getLines()(0x9158d20[WordCounter]) = 59
WordCounter::getAll()(0x9158d20[WordCounter])
    = 0x9f6093e8[Triple]
```

- Traces (including data structures) are our unifying lens.
- Wanted: described conversions between these traces...
A lot of code is (can be) compiled down to object code.

**Initialise WC Module**
**Allocate WC Instance** $\mapsto o$
**Initialise WC Instance**$(o)$
**GetWords**$(o)$
**GetCharacters**$(o)$
**GetLines**$(o)$
**GetAll**$(o) \mapsto (w, l, c)$
**Finalise WC Instance**$(o)$
**Deallocate WC Instance**$(o)$
**Finalise WC Module**

- Traces (including data structures) are our unifying lens.
- Wanted: described conversions between these traces…
- …and a “more unifying” abstract form
To capture styles, we extend the Cake linking language:

- a rule-based language for describing adapters
- declarative
- black-box, convenient, …

Existing Cake compositions…
To capture styles, we extend the Cake linking language:

- a rule-based language for describing adapters
- declarative
- black-box, convenient, ...

...cf. with styles:
Cake code consists of rules which *relate* interface elements.

```plaintext
interfaceA ←→ interfaceB
{
   my_function(arg1, arg2) → your_function(arg2, arg1, 42);
}

values MyStructure ←→ YourStructure {
   foo ←→ bar;  // corresponding fields
   /*  ...  */
}  // relational equiv. of "struct", "class" etc.

table myEnum ←→ yourEnum {
   BEER ←→ GROG;  // corresponding constants
}  // relational equiv. of "enum" etc.
}
```

Not shown: many-to-many rules, context-sensitive rules . . .
Styles relate *more concrete with more abstract* interfaces.

```c
style c89_style_booleans {
    0    ➞ false ;
    _    ➞ true ;
    1    ← true ;
};
style shell_style_booleans {
    0    ➞ true ;
    _    ➞ false ;
    1    ← false ;
};
```

Styles are applied when declaring a pre-existing component.

```c
exists c89_style_booleans(elf_reloc(”componentA.o“)) componentA;
exists shell_style_booleans(elf_reloc(”componentB.o“)) componentB;
```
How it works

```plaintext
abstraction

in style
c89_style_booleans
(BOOL)

in empty style

BOOL
false
0
related vertically by style rules

style-enabled high-level matching

compiler chooses the most abstracting flow

in style
shell_style_booleans
(BOOL)

in empty style

int
0=false false 1

low-level conversion would ignore the latent boolean abstraction
```

Composing... – p.33/68
Conclusions & work in progress

Extra stuff in the paper:

- a more complex example (JNI dispatch)
- composition of styles
- preliminary styles survey (in Appendix) – more needed!

No implementation yet!

- Cake compiler includes some foundations
- background project for me (collaborators welcome!)

Any questions on this bit?
Outline

- the Cake language
- interface styles
- DwarfPython
- interface hiding
Programming languages... 

Programming languages are great, but... 

- there’s lots of them! 
- each has good and bad points 

Goal: make language a per-function design choice... 

- ... not an obtrusive, world-changing decision 
- Problem 1: mixed-language (“foreign code”) is costly 
- Problem 2: need tool support (debuggers, profilers, ...) 

Most languages are implemented on VMs... 

- designed for one or few languages 
- “obtrusive”: “own” a process, own toolset, reinvent OS
static PyObject* Buf_new(
    PyTypeObject* type, PyObject* args, PyObject* kwds) {
    BufferWrap* self;
    self = (BufferWrap*)type->tp_alloc (type, 0);
    if (self != NULL) {
        self->b = new_buffer();
        if (self->b == NULL) {
            Py_XDECREF(self);
            return NULL;
        }
        Py_DECREF(self);
        return NULL;
    }
    return (PyObject*)self;
}
Unifying infrastructures help

“Isn’t this already solved?”

- JVM, CLR et al. unify many languages...
- “unify”ing FFI and debugging issues

But we could do better:

- what about *native* code? C, C++, ...
- not all languages available on all VMs
- ... FFI coding is still a big issue

What’s the “most unifying” infrastructure?
What’s in a virtual machine?

A virtual machine comprises...

- support for language implementors
  - GCing allocator; interpreter/JIT of some kind
  - object model: “typed”, flat ...
  - ... on heap only

- support for end programmers, coding
  - core runtime library (e.g. reflection, loader, ...)
  - “native interface” / FFI

- support for end programmers, debugging / “reasoning”
  - interfaces for debuggers, ...

- support for users / admins (security, res. man’t, ...)

Composing... – p.39/68
What’s in a virtual machine? an OS process + minimal libc?

The “null” virtual machine comprises…

- support for language implementors
  - GCing allocator; interpreter/JIT of some kind
  - object model: “typed”, flat opaque…
  - … on heap only or stack or bss/rodata

- support for end programmers, coding
  - core runtime library (e.g. reflection, loader, …)
  - “native interface” / FI

- support for end programmers, debugging / “reasoning”
  - interfaces for debuggers, … at whole process scale

- support for users / admins (security, res. man’t, …)
Embracing and extending the “null VM”

For most omissions, we can plug in libraries:

- JIT/interpreter...
- choose a GC (Boehm for now; can do better...)

What about reflection?

- ... more generally, “dynamic” features

Debugging infrastructure supports all kinds of dynamism:

- name resolution, dynamic dispatch, ...
- object schema updates (with some work)

... on compiled code, in any (compiled) language!
Well, almost...

Building “null VM” Python means plugging a few holes:

- ... that are *already* problems for debuggers!
- that fit neatly into runtime and/or debugger facilities

I’m going to focus on a specific “hole”.

- For the rest, ask me (or trust me... )
Some equivalences

<table>
<thead>
<tr>
<th>debugging-speak</th>
<th>runtime-speak</th>
</tr>
</thead>
<tbody>
<tr>
<td>backtrace</td>
<td>stack unwinding</td>
</tr>
<tr>
<td>state inspection</td>
<td>reflection</td>
</tr>
<tr>
<td>memory leak detection</td>
<td>garbage collection</td>
</tr>
<tr>
<td>altered execution</td>
<td>eval function</td>
</tr>
<tr>
<td>edit-and-continue</td>
<td>dynamic software update</td>
</tr>
<tr>
<td>breakpoint</td>
<td>dynamic weaving</td>
</tr>
<tr>
<td>bounds checking</td>
<td>(spatial) memory safety</td>
</tr>
</tbody>
</table>

For each pair, implement using the same infrastructure...
DwarfPython is an implementation of Python which

■ uses DWARF debug info to understand native all code
■ unifies Python object model with native (general) model
■ small, uniform changes allow gdb, valgrind, …
■ deals with other subtleties…
  ♦ I count 19 “somewhat interesting” design points

Not (yet): parallel / high-perf., Python libraries, …
Implementation tetris (1)

- C library
- native libs
- Python code
- hand- or tool-generated FFI-based wrapper code
- CPython or similar implementation

operating system
instruction set architecture

Composing... – p.44/68
Jython or similar implementation

Python code

VM libs

VM

native libs

C library

operating system

instruction set architecture

some native libraries inaccessible from Python
Objects are not really opaque…

```python
>>> import ellipse  # dlopen()s libellipse.so
>>> my_ellipse = native_new_ellipse()
>>> print my_ellipse

Invariant 1: all objects have DWARF layout descriptions…
```
>>> import c # libc.so already loaded
>>> def bye(): print "Goodbye, world!"
...
>>> atexit(bye)

Invariant 2: all functions have ≥ 1 “native” entry point

- for Python code these are generated at run time

DwarfPython uses libffi to implement all calls
Dynamic dispatch means finding object metadata. Problem!

Native objects are trees; no descriptive headers, whereas...

VM-style objects: “no interior pointers” + custom headers
Wanted: fast metadata lookup

How can we locate an object’s DWARF info

- . . . without object headers?
- . . . given possibly an interior pointer?

Solution:

- is object on stack, heap or bss/rodata? ask memory map
- if static or stack, just use debug info (+ stack walker)

In the heap (difficult) case:

- we’ll need some malloc() hooks . . .
- . . . and a memtable.
  - read: efficient address-keyed associative structure
Inspired by free chunk binning in Doug Lea’s (old) malloc.
Inspired by free chunk binning in Doug Lea’s (old) malloc.

As well as indexing *free* chunks binned by *size*,
... index *allocated* chunks binned by *address*
How many bins?

Each bin is a linked list of chunks

- thread next/prev pointers through allocated chunks...
  - hook can add space, if no spare bits
- also store allocation site (key to DWARF info)
- can compress all this quite small (48 bits)

Q: How big should we make the bin index?
A: As big as we can!

- given an interior pointer, finding chunk is $O(bin\text{size})$

Q: How big can we make the bin index?
A: Really really huge!
Really, how big?

Exploit

- sparseness of address space usage
- lazy memory commit on “modern OSes” (Linux)

Bin index resembles a linear page table.

After some tuning...

- 32-bit AS requires $2^{22}$ bytes of VAS for bin index
- covering $n$-bit AS requires $2^{n-10}$-byte bin index...
- use bigger index for smaller expected bin size
What’s the benefit?

Faster and more space-efficient than a hash table

■ also better cache and demand-paging behaviour?

Some preliminary figures (timed gcc, 3 runs):

■ gcc uninstrumented: 1.70, 1.76, 1.72
■ gcc + no-op hooks: 1.73, 1.76, 1.72
■ gcc + vgHash index: 1.83, 1.82, 1.85
■ gcc + memtable index: 1.77, 1.78, 1.77

Memtables are not limited to this application!

■ e.g. Cake “corresponding objects” look-up
■ ... your idea here
Status of DwarfPython

Done: first-pass simplified implementation

- DWARF-based foreign function access
- no dynamic lang. features, debugger support, …

Full implementation in progress…

- including proof-of-concept extension of LLDB
- + feedback into DWARF standards!
Calling native functions:

- instantiate the data types the function expects
- call using `libffi`

In Parathon, an earlier effort, we had:

```cpp
ParathonValue* FunctionCall::evaluate(ParathonContext& c)
{
    return call_function (this->base_phrase->evaluate(c),
                         /* invokes libffi */ this->parameter_list->asArgs(c));
}
```

Now we have:

```cpp
val FunctionCall::evaluate() // <-- only context is the *process* i.e. stack
{
    return call_function (this->base_phrase->evaluate(),
                          this->parameter_list->asArgs());
}
```

The interpreter context is the process context!
Primitive values

objects

value-unique objects

singleton objects

immutable objects

in this region, object references are interchangeable with values
Out-of-band metadata

Traditional approach: in-band headers point to object metadata

DwarfPython approach: metadata kept out-of-band and looked up associatively

Object metadata

- Class point:
  - Field: x ...
  - Field: y ...

- Class ellipse:
  - Field: min ...
  - Field: maj ...
  - Field: ctr ...
  - Method: draw
  - Method: move

Object data

- Ell centre:
  - X: -1
  - Y: 8

Look-up function

- Address: 0x00c0ffee
- Address: 0xdabadbeef

- Ell centre:
  - X: -1
  - Y: 8
the Cake language
interface styles
DwarfPython
interface hiding
There’s still something about software…

Dealing with change is hard work.

- *porting*
- *wrapping*

New languages are hard to adopt

- each language is its own silo

Usually we eschew *integration*:

- rewrite from scratch
- single-language programming
The idea in one slide

Parnas pioneered *information hiding*.

- interface changes are painful, so . . .
- keep interfaces *minimal*, to avoid change

But interfaces *do* change. What can be done?

- radically separate integration: *interface hiding*
  - Error: “import” statement deprecated
  - instead, *declare* (meaning *invent*) target interface
- specialized tool support for integration
  - complementary languages (Cake is one example)
Separate integration

Hardware (and other domains)

- chip *invents its view* on outside
- keeps components simple
- . . . and composable
- . . . and cheap

By separating integration . . .

- *physically* (modules)
- *notationally* (languages)

Result: components are *less complex* and *less coupled*. 
decoder = mpeg2_init();
info = mpeg2_info(decoder);

**do** { state = mpeg2_parse(decoder);

```
switch (state) {
    case STATE_BUFFER:
        size = fread (buffer, /* ... */);
        mpeg2_buffer (decoder, buffer, /* ... */);
        break;
    case STATE_SLICE:
        for (i = 0; i < seq->luma_height; i++)
            fwrite (info->display_fbuf-> /* ... */);
        break;
}
```

} **while** (size);

mpeg2_close (decoder);
avcodec_init(); av_register_all();
codec = avcodec_find_decoder(codec_id);
ap−>time_base= (AVRational){1, 25};
ap−>pix_fmt = PIX_FMT_NONE;
err = av_find_stream_info (ic);
for (i = 0; i < ic−>nb_streams; i++) {
    AVCodecContext *enc = ic−>streams[i]−>codec;
    if (enc−>codec_type == CODEC_TYPE_VIDEO)
        video_index = i; }
c = ic−>streams[video_index]−>codec;
for (;;) { while (pkt−>size > 0) {
    picture = avcodec_alloc_frame();
    len = avcodec_decode_video2(c, picture, &got_picture, pkt);
    if (len >= 0 && got_picture) {
        for (i=0; i < c−>height; i++) /* for each row */
Since no objections were raised in kde-core-devel, I am merging the kdelibs4-dbus branch back into trunk. KDELibs compiles, links and installs with this, but obviously all other modules will fail to build. Let the porting commence.

CCMAIL:kde-core-devel@kde.org, kde-buildsystem@kde.org
## Preliminary case: table X

<table>
<thead>
<tr>
<th></th>
<th>mpeg2</th>
<th>ffmpeg</th>
<th>db</th>
<th>sqlite</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>API function count</strong></td>
<td>24</td>
<td>151</td>
<td>208</td>
<td>214</td>
</tr>
<tr>
<td><strong>mean signature size</strong></td>
<td>3.3</td>
<td>4.3</td>
<td>2.8</td>
<td>4.3</td>
</tr>
<tr>
<td><strong>API structures count</strong></td>
<td>10</td>
<td>26</td>
<td>25</td>
<td>12</td>
</tr>
<tr>
<td><strong>mean structure size</strong></td>
<td>5.6</td>
<td>19</td>
<td>17</td>
<td>8.0</td>
</tr>
<tr>
<td><strong>slice function count</strong></td>
<td>5</td>
<td>14</td>
<td>5</td>
<td>12</td>
</tr>
<tr>
<td><strong>mean signature size</strong></td>
<td>2.8</td>
<td>3.1</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td><strong>slice structures count</strong></td>
<td>3</td>
<td>6</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td><strong>client size (LoC)</strong></td>
<td>71</td>
<td>93</td>
<td>51</td>
<td>75</td>
</tr>
<tr>
<td><strong>client API calls</strong></td>
<td>5</td>
<td>14</td>
<td>5</td>
<td>13</td>
</tr>
<tr>
<td><strong>helper calls</strong></td>
<td>2</td>
<td>1</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td><strong>mean call size</strong></td>
<td>3.4</td>
<td>3.2</td>
<td>4.9</td>
<td>3.9</td>
</tr>
</tbody>
</table>
We can practise interface hiding now

- as a consequence of separate compilation
- C and C++: just declare the interface you want
- in general, define a private interface or signature or . . .

But easier with tools like Cake (and successors!). Idea:

- Can we take some direct-programmed code . . .
- . . . and factor out concrete API details automatically?
- Like factoring out an abstraction layer . . .
- . . . but written in Cake!

Also want a survey of API diversity . . .
Conclusions

Interface hiding is a radical “next step” in modularity.

- reduce component complexity
- minimises coupling
- retain composability with many libraries

Integration domains abstract integration

- better notational abstractions for integration
- absorb change; distance from specific infrastructure; . . .
- incremental adoption is feasible

Thanks for your attention. Any questions?