Accessible Language-Based Environments of Recursive Theories*

(a white-paper advocating widespread, unreasonable behaviour)

Ian Piumarta
piumarta@speakeasy.net

2005-09-30†

> "The reasonable man adapts himself to the world: the unreasonable man persists in trying to adapt the world to himself. Therefore all progress depends on the unreasonable man."
>
> — George Bernard Shaw, *Maxims for Revolutionists*

*ALBERT is a publically-diffusable working title for the system described in this whitepaper while we think of less (legally) troublesome replacements for 'pepsi' and 'coke'.

†This is a work-in-progress and should not be considered final. Repetition is rampant and parts may be incoherent, incomplete, or internally inconsistent.

Contents

1 Introduction **3**

2 Conventional programming languages **4**

3 Combined Object-Lambda Architectures (COLAs) **6**
3.1 End-user abstractions . . . 6
3.2 Modularity and reuse . . . 8
3.3 Construction and operation . . . 8

4 Representation: objects and messaging **11**
4.1 Object format . . . 11
4.2 Methods and messaging . . . 13
4.3 Interoperability . . . 14

5 Behaviour: symbolic expressions and transformations **15**
5.1 Structural transformations . . . 16

6 Circular implementation **18**
6.1 Bootstrapping . . . 19

7 Unconventional programming languages **22**

8 Conclusion **24**
8.1 Omissions . . . 24
8.2 Status . . . 24
8.3 Perspectives . . . 25

1 Introduction

A new way to construct programming languages, systems, environments and applications (herein collectively referred to as 'systems') is described. The priorities are somewhat different from those of conventional systems, emphasising (above all else) simplicity, openness, evolution, and pervasive user-centred implementation. The goal is that users be able to understand and modify any part of the system, and that the system be organised in a way that encourages them to do so.

- **Simplicity** — the system should be simple to understand and to modify. *Self-similarity* is pervasive, both globally (the system entirely describes and implements itself, within a homogeneous object-oriented paradigm) and locally (each stage in the implementation chain differs from those adjacent in a single aspect according to a transformation whose interface is as simple as possible, but no simpler).
- **Openness** — no part of the system is hidden. All stages are visible, accessible to and modifiable by, the user. The transformations between adjacent stages are evident.[1]
- **Evolutionary programming** — the system supports and encourages evolutionary approaches to design, implementation and maintenance. The basic system is designed to be evolved from within to create a final end-user system. Fluid semantic transformations in the implementation encourage programmers to evolve personally, inventing and applying more appropriate and expressive paradigms as they learn new techniques and more about the solution spaces in which they can work. Dynamic, pervasively late-bound, implementation supports rapid adaptation to changing external conventions and constraints.
- **User-centred construction** — the system strives to serve the programmer (and not the other way around). The programmer can take control of (and responsibility for) all aspects of their programming language(s), system and environment.

In the remainder of this document, each section begins with a short summary of its contents in a box, like this. Section 2 describes the current state of much of the art of conventional programming systems. Section 3 describes a new approach to system building, with detailed explanations of the representation of syntactic structures and the manipulations performed on them in Sections 4 and 5, respectively. Section 6 shows how the system implements itself, from top to bottom, and Section 7 contrasts the resulting unconventional systems from those introduced earlier (in Section 2). Section 8 wraps it up, describing where it's at and where it's going.

[1] It has been noted that: "The best way to predict the future is to invent it." We take this further, as the *Open Future Principle*: "The best way to implement the future is to avoid having to predict it." Consequently, nothing whatsoever in the system or its implementation is early-bound.

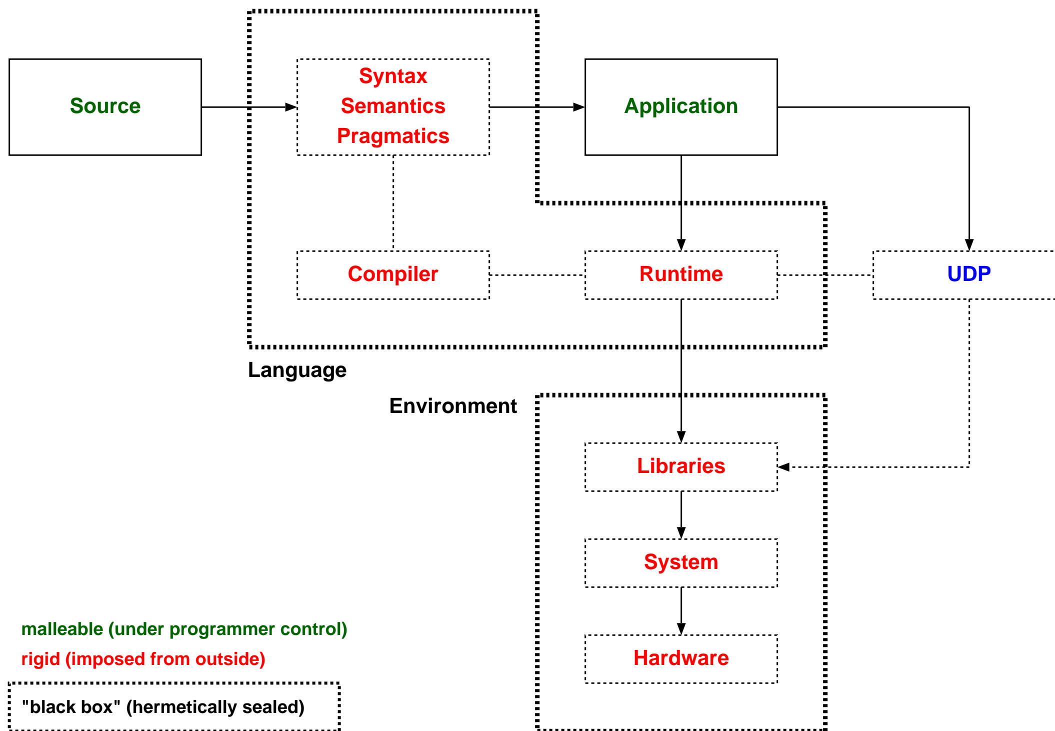

Figure 1: Conventional programming languages. In green: the programmer can manipulate the source code at will, and has some illusion (to a greater or lesser extent) of implicit control over the executable application generated from it. The red boxes are imposed on the programmer from outside and include all aspects of the language and all (or most) of the environment; usually inaccessible from user-level code, they form two impenetrable black boxes.

---

2 Conventional programming languages

> The programmer has total control over the source code (within the limits of syntactic correctness), and some illusion of control (indirect, by implication only) over the final application that is generated. All other aspects of the system are rigidly predetermined and present (for all but the simplest programs and/or most simple-minded programmers) impenetrable, often infuriating, *artificial* and *arbitrary* barriers to creativity, expression, and the use of the most elegant and appropriate solutions.

Figure 1 illustrates a typical programming language.[2]

Source code is submitted to a compiler that creates a corresponding executable application. When run, the application (presumably) performs some useful interaction with its external environment by invoking functions in the system libraries and/or OS. The programmer is dealing with two impenetrable, hermetically-sealed black boxes: the *language* and the

[2]In addition to the compiler, the 'system' probably consists of nothing more than a text editor, some kind of symbolic debugging technique, and maybe an offline cross-reference index generator.

environment.[3]

The language is a combination of syntax (restricting the legal content of the source code), semantics (placing predefined meaning on that content), and pragmatics (the range of externally-visible effects that are possible during execution). All three of these tend to be rigid (designed by committee, for generality rather than fitness for any particular purpose), inaccessible to the programmer (concealed in a hermetically-sealed 'language' box), and presented as *faits accomplis* to some unfortunate loser community.

With few exceptions, the environment (libraries, OS) is accessible only through the facilities provided by the runtime support (probably designed by the same committee responsible for the language). Accessing obscure or nonstandard facilities is either impossible, inefficient, or profoundly disruptive to the creative programming process.

Languages that do provide access to nonstandard facilities often do so through a foreign function interface (FFI) mechanism (matching impedances for calls out to the system) and/or a user-defined primitive (UDP) mechanism (in which hand-written C code performs the same kind of impedance matching and call-out). FFIs are expensive (normally involving explicit synthesis of a dummy stack frame for each call-out) and their acceptability depends on the ratios of internal processing and external workload performed per call-out. UDPs demand specialised knowledge from the programmer (the ability to shift a level of abstraction — and usually representation — within the implementation hierarchy) and are prohibitively disruptive for exploratory and high-availability systems (edit, recompile, stop and re-launch the world). Both approaches are limited to pragmatics (environmental effects); no specialisation of semantics is possible. Modification of the language (syntax, semantics) demands knowledge even more specialised than that needed to add primitives, and is in the domain of far fewer programmers. (It also assumes availability of the language implementation for inspection, modification and recompilation.)

[3] In the case of C[++] it's really one and a half black boxes, since parts of the environment (the libraries) are written in the same language as (or at least are ABI-compatible with) that which the programmer is using.

3 Combined Object-Lambda Architectures (COLAs)

A COLA is a pair of mutually-sustaining abstractions. One provides representation and the other (behavioural) meaning. A minimum of each is imposed: the most desirable end-user structures and abstractions are not the goal, but rather the simplest that can produce a fully self-describing (and self-implementing) system.

Representation is provided by prototype-like objects exchanging messages, organised into clone families (somewhere between lightweight classes and instance-specific behaviour); the semantics of messaging are defined recursively, by sending messages to objects. Meaning is imposed on these representations by transforms that convert structures (similar to symbolic expressions in the lambda calculus) into executable forms; the semantics of structures are defined recursively, by representing transforms as structures indistinguishable from those that they transform. One such executable form provides the implementation of methods installed in the objects of the representation; the overall implementation is circular.

The implementation language and abstractions of the system are precisely the language and abstractions that the system implements. Providing a dynamic execution model, for which both dynamic and static code can be generated, eliminates the need for a central interpreter- or VM-like agent, and ensures that everything (including the deepest 'kernel' behaviour) can be modified, dynamically, on-the-fly.

A self-hosting COLA (Combined Object-Lambda Architecture) is a radically different approach to building programming languages, systems and applications.[4] A COLA is named after the two abstractions in its implementation; these will be described below. First however, a brief description of the end-user view of a resulting system will be useful.

3.1 End-user abstractions

A COLA-based dynamic programming system is illustrated in Figure 2, consisting of a hierarchy of pluggable stages:

- A **front end** acquires text (or some other unstructured form) from an input device (console, file, network, ...).
- A **parser** converts this text into a structured form corresponding to an abstract syntax tree (AST).
- The **tree compiler** 'walks' the ASTs, repeatedly applying transformations to the trees until nothing remains.[5] The 'output' from this stage is a sequence of *abstract instructions* describing a complete function.[6]

[4]The 'representation' layer provides a language that is similar to one desirable end-user language, but is not an ideal (pervasively late-bound) implementation of that language. It is code-named 'Pepsi'. The 'meaning' layer provides everything required for a pervasively late-bound implementation of Pepsi. Since this is 'the real thing', it is (of course) code-named 'Coke'. (Readers under the age of 35, or those never significantly exposed to an anglo-saxon culture, might have trouble figuring this one out.)

[5]Transforming a node within the AST can be a simple rewrite (like a macro), or an imperative action (modifying compiler state and/or causing code to be generated) that results in another AST or nothing (when any side effect has completely consumed the meaning of the AST).

[6]Abstract instructions share representation with every other object/structure in the system.

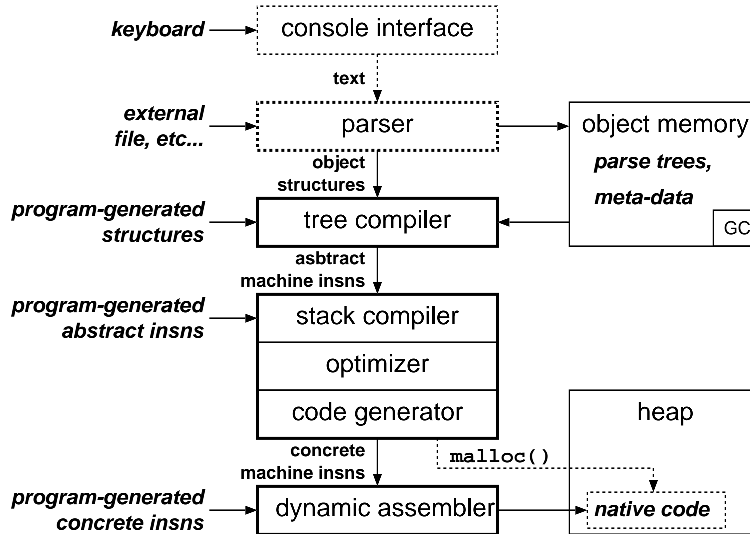

Figure 2: Logical architecture. The representation layer (objects and messaging) provides state and behaviour for the object memory, storing structures for input (parse trees), intermediate representations, and per-session persistent state (akin to meta data). The vertical compilation chain, potentially ranging from an interactive 'console' through to the assembly of native code instructions, is implemented as a series of transformations on semantic structures stored in the object memory, reified as methods (or functions) associated directly (or indirectly) with objects and structures. The transformations themselves are described by structures in the object memory. Each component in the chain (assembler, stack-oriented architecture-neutral code generator, structure/tree compiler, etc., are libraries that depend only on lower components within the chain. Each is a useful entry point into the compilation chain and can be used independently (inputs driven by some external program) or within a deeper chain (inputs driven by the component 'above' it in the chain) or as a point to rejoin the chain when higher components have been modified from their original form *in-situ* by the application.

- The **virtual processor** translates abstract instructions into native instructions, by a further process of transformation. A few carefully chosen optimisations are applied during this stage.[7]

- Finally, a **dynamic assembler** converts native instructions into binary for execution. (In this dynamic, interactive, incremental example the generated code is executed immediately for its side effects.)

The presentation of each stage is as simple as possible, typically a single object (the only 'client-side' state retained by a superjacent stage) whose methods define a functional interface. Reconfiguration of the implementation chain is achieved by presenting any compatible object to its 'client' stage. Alternate stages include interpreted execution strategies (of tree structures or abstract instructions), generation of symbolic code (retention of tree structures

[7]The criterion of choice is that each optimisation be fast while contributing significantly to the efficiency of the final code; i.e., it has a small price-performance ratio.

as an executable form, bytecodes, etc.), assembly to static (externally-stored executable) code rather than dynamic code, and so on.

3.2 Modularity and reuse

Each stage provides a useful service in its own right and is organised to allow compilation to a static, standalone, C- or C++-compatible library. This is trivial for the assembler and virtual processor due to their allocation behaviour.

The intrinsic object system is largely hidden from the average user, and consists of a handful of 'fundamental' objects.[8] Memory management (allocation, deallocation, garbage collection) is implemented (like most everything else) as methods installed in objects. The _object prototype, root of all object hierarchies, implements a 'null' memory management policy: allocation (and explicit deallocation) is from the C heap, and out-of-memory conditions are fatal. It is the responsibility of hierarchies delegating to _object to override this behaviour with something more useful (if required).

The dynamic assembler is a single object (with no aggregate internal state). Once allocated, there is no need ever to free it (thread-local use notwithstanding). Automatic memory management is irrelevant, and the trivial implementation inherited from _object is sufficient.

A virtual processor has no persistent internal state, does not export state to the client, and deals with one function at a time. Allocation within the VPU is monotonic; once translation of a function is complete, all internal state is released atomically.[9] Trivial wrapper methods provide monotonic object allocation within blocks obtained from the heap for all internal VPU state, followed by explicit deallocation of the blocks *en masse*.

The tree compiler is the next independently useful stage, and is effectively the unique entry point into more abstract stages of a COLA. (Parsers and front ends ultimately manipulate tree compiler objects.) Some of its objects persist, and are subject to automatic memory management.[10]

3.3 Construction and operation

Figure 3 shows the implementation of a COLA.

On the left is the simplest possible 'pure' object-oriented language: just the sufficient and necessary elements to represent and interact with structure.[11] The only implicit operation is dynamic binding in the method cache (an optimisation, *not* an operation that defines semantics). Correspondingly, dynamic binding (an operation that does define semantics) is *not* an 'intrinsic' (or 'primitive') operation, and occurs explicitly whenever the method cache misses; a (real) message is sent to a (real) object to perform (define the semantics of) method lookup.

Messaging is therefore self-describing (the semantics of sending messages to objects are described and implemented by sending messages to objects). Additional mechanisms (novel binding — on multiple arguments, for example — including delegation/inheritance or other

[8] A common _object prototype (the parent of all other objects), along with the prototypes and families required to give it trivial dynamic behaviour: _selector, _implementation, _binding, and _vtable.

[9] Similar to the monotonic allocation and bulk deallocation in 'ObStack's.

[10] Any number of interchangeable strategies are possible, according to the needs of the application in its deployment context.

[11] Local structure is imposed to meet the criterion that the system can have any global structure.

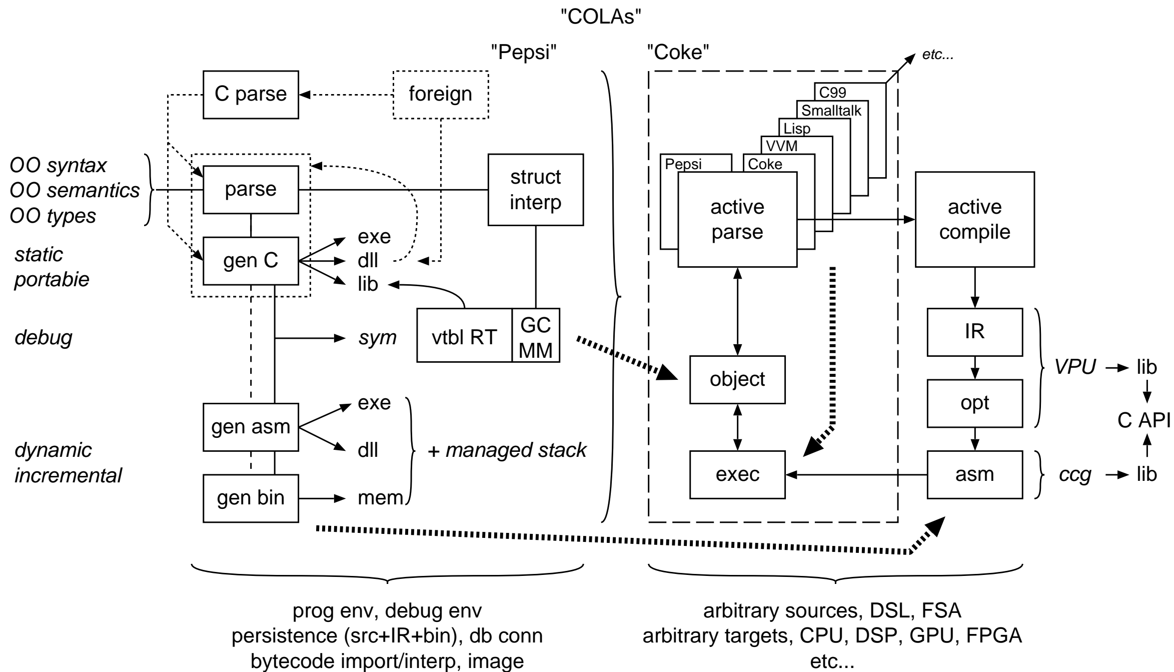

Figure 3: Architecture of a COLA. **On the left**, a minimal 'pure' object-oriented language provides representation and behaviour of data structures, whose execution model (and corresponding runtime support) is rigid only in the intrinsic vtable layout and a global method cache (see text). Back ends can generate code in a variety of ways: portably and indirectly (generating C as a high-level assembly language) with little or inefficient debugging support, or directly to static (file-based) or dynamic (in-memory) native code, with full symbolic debugging. Additional services include foreign language importers (e.g., a C99 parser to extract platform data types and interfaces), execution of semi-compiled code (interpretation of data structures without translation to native code), and compilation for managed (rather than hardware-native) stack models. **On the right**, the data structures represent (dynamic) behaviour for all phases of program translation: parsing, parsed representation, compilation (including intermediate representations), optimisation, and code generation. ('Dynamic' refers to late-bound behaviour throughout the architecture that can be modified by users as desired.)

---

reuse mechanisms) are implemented by overriding the default methods that implement dynamic binding.

All manipulation of objects is accomplished by message passing. All runtime structures (selectors, virtual tables, etc.) are real objects. The contents (or 'shape') of an object are defined functionally, by the methods that access its state.

The above gives us a complete 'theory' (or 'algebra') of communication between objects (messaging between objects from within method activations); it will be described in more detail in Section 4. It does not give us a 'theory' ('algebra') of meaning (behaviour) to describe the internal implementation of methods.

On the right of Figure 3, structures (built from the above objects) are formed into a forest of ASTs. Each successive AST in the forest is evaluated, by compiling it and then

executing any resulting form. (The entire AST evaluation chain — from front-ends, through IR, optimisations, back ends and code generators, to assemblers — is implemented within this one object system.)

The meaning of each AST node is given by its association (a dynamic binding, determined during the evaluation of the AST) with a compilation closure (implementing syntactic, semantic or pragmatic meaning for the AST). A syntactic closure produces a rewritten AST (similar to Lisp macros); semantic/pragmatic closures produce full or partial implementations of their AST, generating IR and/or accessing back end/assembler features as required. Either can have arbitrary side effects, and neither need produce a runtime effect (for structures whose meaning is limited to compile-time effects). These closures are just methods executing in the AST, or some associated object.

Pluggable back ends can be dynamic (producing native code in memory), symbolic (producing bytecode), static (producing assembler source for a static executable), or any intermediate point along a continuum spanning the above.[12] Generated static/dynamic native code complies (unless otherwise overridden by a 'pragmatic' compilation closure) with the local C ABI (ensuring seamless integration with OS, libraries, other languages, etc. — or anything compatible with C calling conventions).

Front ends can be for any input language (textual or structured). One particular front end describes the intrinsic object language of the system.

We now have self-describing theories of both objects (and messaging) and the semantics of messaging and non-messaging 'primitive' operations within methods.[13] Moreover, these descriptions are self-implementing and (whenever necessary) dynamically self-modifying.

Many interesting and useful systems (based on existing languages or otherwise, dynamic or static, OOP or not, etc.) can be constructed by 'mutating' the appropriate elements of the above implementation chain into the desired target system.

Any combination of static and dynamic language features and deployment techniques are possible. Virtual machines can be replaced by a set of 'kernel' object types (including everything required to implement the compilation chain) that are compiled into a static (stored on some persistent media for later execution) executable. The single paradigm for both static and dynamic code allows any parts of this statically-generated kernel code to be modified dynamically at runtime.[14]

[12] In general, dynamic code affects the programming environment and static code affects the (offline) application being compiled. Many applications however might permanently simply coexist, in dynamic form, with the COLA that supports them.

[13] The implementation (language) model is (precisely) the implemented (language) model.

[14] Statically-compiled COLA code should be thought of as 'frozen' dynamic code. When such code is run, the system's execution state is created incrementally as if the top-level declarations and statements in the program had been read, compiled and executed interactively.

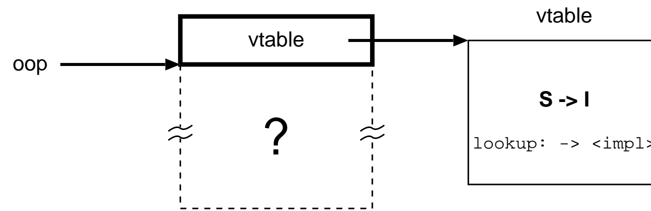

Figure 4: Intrinsic dynamic behaviour. The runtime support implements dynamic binding (message dispatch) in the simplest possible manner. Each object has an (indirectly or directly) associated virtual table that maps message identifiers (selectors) to implementations. Object behaviour is therefore late-bound (according to the contents of the virtual table) but the format of the vtable is not, since the runtime must be capable of interpreting the contents of the virtual table to find the method implementation associated with a particular selector.

---

4 Representation: objects and messaging

> The intrinsic object model is the simplest possible that can support messaging. Binding in a method cache, the only primitive operation provided, does not define the semantics of messaging. The operation that does define messaging semantics, method lookup after a cache miss, is not defined primitively; method lookup is performed by sending real messages to real objects. All structures involved in the implementation of messaging are full objects that respond to messages.
>
> Simplicity breeds generality and flexibility, and this model extends itself dynamically to provide more useful abstractions (such as delegation for behavioural reuse). The in-memory format of objects is designed such that 'wrapping' an ordinary object imported from a foreign language/paradigm (to add dynamic behaviour) preserves identity between local and foreign references.

To preserve generality and flexibility, the intrinsic object model is as simple as possible (too simple to be of much practical use). It is transformed into a usable object model (supporting reuse, composition, etc.) by extending it in terms of itself (a self-similar implementation) as outlined above.

4.1 Object format

Intrinsic dynamic (late-bound) behaviour is associated with an object through (for want of a better word) a virtual table ('vtable').[15] Sending a message to the object consists of finding an implementation (at message send time) within the vtable of the receiver that corresponds to the selector of the message being sent. Figure 4 illustrates this intrinsic 'lookup' operation.

The intrinsic model provides objects and messaging, but no way to add (for example) behavioural composition or reuse (delegation, inheritance, whatever). Consider instead the object just described as a 'binding object', primarily responsible for implementing a 'lookup'

[15] The association of object with its virtual table is unspecified, and can be explicit or implicit. For sake of illustration, assume for now that an object is known by an address in memory at which some corresponding state is stored and that its vtable is identified by a pointer within that state.

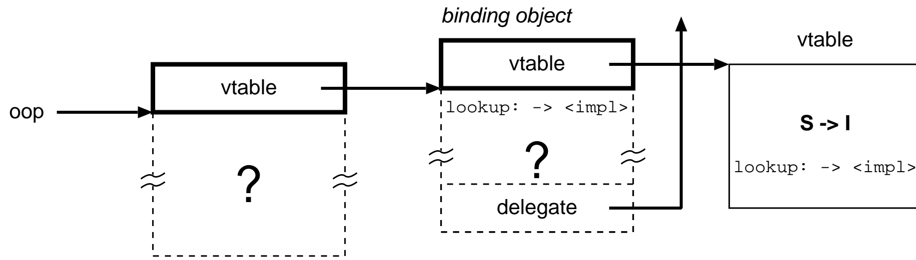

Figure 5: Dynamic dynamic behaviour. More useful behaviour is obtained by replacing the intrinsic vtable object with a user-defined *binding object*. Binding objects are not limited to the intrinsic vtable format and can implement arbitrary lookup operations, following the desired semantics of messaging. In the example shown here the binding object includes, in addition to an unspecified means of associating selectors with implementations, a reference to a *delegate* object that will attempt to field messages that are not understood by the receiver. The details of when and how this delegate object are used are under the control of the lookup operation defined by the binding object.

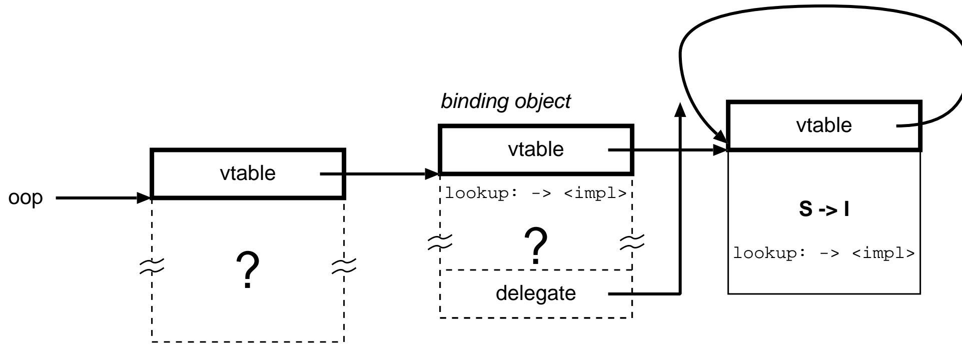

Figure 6: Everything is an object. The intrinsic vtable is its own vtable (the point of circularity in the object system) and is manipulated internally, just like any object, by sending messages to it. The implementations of the associated methods can be overridden as required by a knowledgeable user.

method. Extending the model sideways (by adding a level of indirection) gives an object whose dynamic behaviour (response to message send) is implemented by a user-accessible object (the intrinsic object from above) whose implementation is dynamic (late-bound). Figure 5 illustrates this arrangement.

The binding object can be extended to implement any behavioural composition or reuse mechanism (the figure illustrates single delegation), define any number of coexisting mechanisms, or create new mechanisms (a clone family of binding objects) to complement (or replace) existing ones.

To complete the model, the initial vtable (providing behaviour for binding objects) is made a real object by associating it with a vtable. As shown in Figure 6, the simplest solution is for it to be its own vtable.

4.2 Methods and messaging

Methods are stored as closures. Their function address is the implementation of the method body, and their closed-over state is unspecified.[16] The code compiled for a message send is therefore equivalent to

```
closure ← bind(object, selector, ...) ;
closure.function(object, closure, ...)
```

where 'bind' searches for a cached method implementation, invoking the lookup operation only on cache miss:

```
bind(object, selector, ...) =
    vtbl ← object[-1] ;
    cache[vtbl, selector]
        ? cache[vtbl, selector]
        : vtbl.lookup(object, selector, ...)
```

(where the '.' operator represents a full message send).

Any set of objects associated with a single vtable share identical behaviour. Such a set of objects is called a *clone family*; modifying the behaviour associated with any object in the family modifies the behaviour of all members of the family.

For ordinary objects (those with physical extension in memory) a pointer to the vtable is stored one word before the contents of the object. Two additional vtables are associated *implicitly* with the object at address zero (nil) and with any object having the lowest (rightmost, 0th) bit set (normally used to represent a 'tagged' integer).[17] (These two vtables are not predefined and are initialised explicitly in user-level code during startup. They can be modified, or exchanged with another vtable, at any time.) The full 'bind' operation, including implicit vtable associations, is therefore:

```
bind(object, selector, ...) =
    vtbl ← object == 0 ? vtbl_nil
                       : object & 1 == 1 ? vtbl_fixint
                                         : object[-1] ;
    cache[vtbl, selector] ? cache[vtbl, selector]
                          : cache[vtbl, selector] ← vtbl.lookup(vtbl, selector, ...)
```

[16] This facilitates mixed-mode execution, where a method closure's function might be a shared interpreter loop and its closed-over state a sequence of bytecodes to be interpreted. Such bytecoded methods would be indistinguishable, insofar as their calling conventions are concerned, from methods fully compiled to native code.

[17] This is for compatibility and efficiency, respectively.

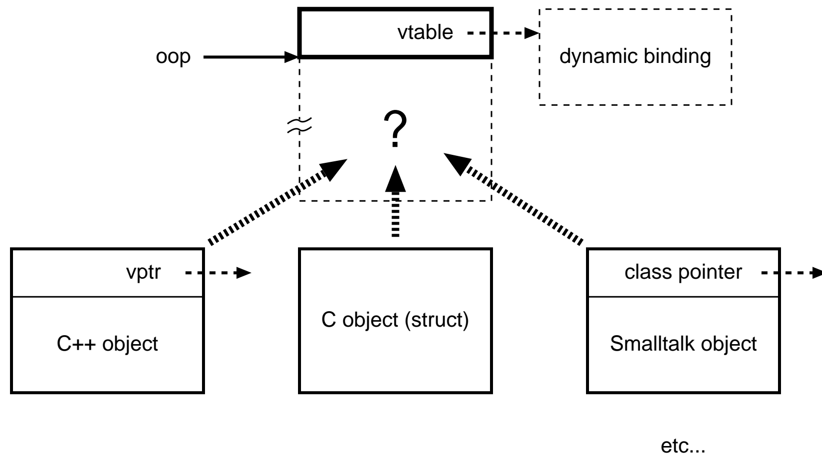

Figure 7: The in-memory representation of objects places the binding object reference one word *before* the address of the object; an object's address therefore refers to the first byte of user data within the object. (The implementation of messaging is transparent to the user.) A consequence of this design is that objects from other languages and paradigms can be encapsulated (as full objects with dynamic behaviour) in a manner that preserves identity between the COLA reference and that within the foreign language/paradigm.

4.3 Interoperability

As shown in Figure 7, the memory format of ordinary objects places a vtable pointer before the address of the object. An 'oop' points to first byte of (user-defined) data in the object body. This is done to facilitate transparent encapsulation of foreign objects, preserving identity between foreign and native references to the object. (This is particularly useful for toll-free bridging to other languages and to library/OS services, in conjunction with parser extensions that process platform header files to extract type information and function signatures for automated interface generation.)

```
eval(nil, context) = nil

eval(tree, context) =
    closure ← transform(type(tree), context) ;
    eval(closure(tree, context), context)

transform(atom, context) =
    (closure ← get-property(context, atom, 'transform)) == nil
        ?  ⟨error⟩
        :  closure

transform(tree, context) = eval(tree, context)
```

Figure 8: Basic transformation engine. A tree is evaluated by applying the 'transform' property (a closure) of the tree's type (an object or structure) to the tree itself. (If the type is a tree, it is first evaluated recursively.) While the transform results in new structure, the resulting structure is evaluated immediately. The leaves of the evaluation graph are either no-ops (evaluating 'nil') or the execution of a transform; all useful side effects therefore occur within the transforms.

---

5 Behaviour: symbolic expressions and transformations

> Objects are composed into syntactic structures representing meaning (behaviour). These structures are translated into an executable form by successive applications of transforms that give semantic meaning to the syntactic structures. These transforms are represented as objects composed into semantic structures. Since a transform is just behaviour, semantic structures are syntactic structures (whose behaviour is applied to syntactic structures); there is no 'meta' level.
>
> Syntactic structures are 'typed'. The most convenient type object is a symbol (a human-readable name) with a 'transform' property associated with the transform appropriate for the structure. Every semantic action/rule in the system can therefore be named, and these names arranged into hierarchical namespaces. Transformations are applied in the context of a particular namespace. Operations on these namespaces (assignment to symbol properties, shadowing in a local namespace, etc.) affects the meaning given to syntactic structures during transformation. The same input structure can mean very different things in different contexts, and users can redefine the semantics of any structure globally or locally to create domain-, application-, or even mood-specific abstractions, paradigms and languages.

The objects of Section 4 are formed into structures representing symbolic *syntactic expressions*, whose meaning (semantics, behaviour, implementation) is described by symbolic *semantic expressions* (also represented as structures composed of the same objects). Syntactic expressions (or structures) are transformed ('evaluated') according to semantic expressions (or structures) by a simple *tree compiler* that in itself places no intrinsic semantic meaning on the structures that it is compiling. This evaluation eventually yields (one, both or neither of) an executable representation in memory (or in a file) and side-effects modifying the evaluation/compilation context of the system itself.

```
eval(nil, context) = nil

eval(atom, context) = emit-value(context, atom)

eval(tree, context) =
    closure ← transform(type(tree), context) ;
    eval(closure(tree, context), context)

transform(atom, context) =
    (closure ← get-property(context, atom, 'transform)) == nil
        ?  apply
        :  closure

transform(tree, context) = eval(tree, context)

apply(tree, context) =
    emit-apply(context,
               eval(type(tree), context),
               map(eval, children(tree), context))

emit-value(context, object) = ⟨emit abstract insns⟩ ; nil

emit-apply(context, func, args) = ⟨emit abstract insns⟩ ; nil
```

Figure 9: Extended transformation engine. Evaluating an atomic object yields a sequence of abstract instructions (via 'emit') generating the implied value. Evaluating a tree involves re-applying (at compile time) the semantic transform (implemented as a closure) associated with the type of the tree to the tree itself, for as many iterations as this process continues to yield a transform. Any remaining structure is re-evaluated. The 'get-property' operation finds the transform associated with an object within the given context. (Humble apologies for the abuse of 'map' but hopefully you get the idea, and it's clearer than the correct 'map(rcurry(eval, context), children(tree))'.)

5.1 Structural transformations

The tree compiler is a simple engine that drives the transformations on (and implied by) object structures. The crux of the engine is shown in Figure 8.

A convenient convention is to place transform properties on symbols, giving each tree a human-readable 'semantic type'. These symbols can be arranged into hierarchical namespaces, such that transforms can be shared (inherited from outer namespaces) or local (overriding similarly-named transforms in outer namespaces).[18]

An AST is therefore a combination of a type symbol (or expression) and zero or more children, where property lists attached to symbols give semantic meaning to (define the transformations applied to) ASTs. Retrieving a property attached to a symbol according to its name is isomorphic to binding a method to an object according to its selector. Each distinct symbol that can denote the type of a tree is created in a singleton clone family with its

[18]This arrangement is particularly useful in the presence of a parser that generates tree structures directly from Lisp-like prefix expressions.

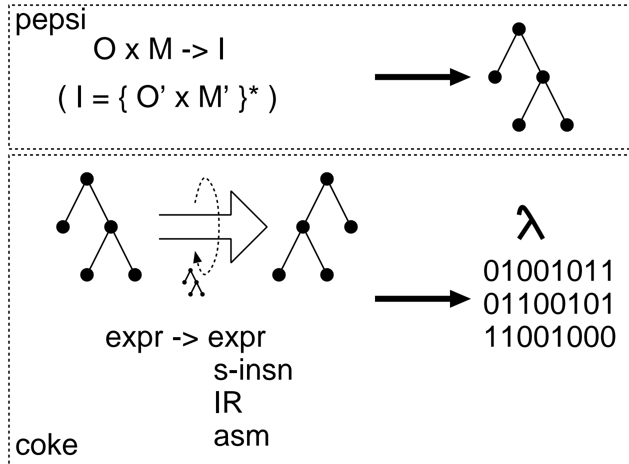

Figure 10: Two mutually-sustaining levels of circular implementation. A representational substrate (shown at the top) provides objects (for structure) and messaging (for interaction with the structure). The messaging model assumes only that some combination of object O and received message M identifies a unique method implementation I that in turn causes zero or more messages M' to be sent to object(s) O'. Syntactic structures (shown at the bottom) are formed to express behaviour (meaning). These expressions are transformed progressively, according to rules that are (naturally) themselves expressed as semantic structures, to produce executable code. The abstractions traversed during these transformations correspond to the phases or passes (analyses, intermediate representations, final assembly) of a traditional compiler.

---

transform installed as a method in the its family's vtable. Applying a transform to a tree is therefore equivalent to (and implemented as) sending the tree's type a message with selector 'transform', passing the tree and its compilation context as arguments.

The engine shown in Figure 8 is cumbersome in that every literal value and identifier in the AST must be placed in a tree whose type identifies it as an r-value to be emitted. Similarly, each function application requires an explicit 'apply' node in the tree. It is convenient to extend the engine slightly to handle these two cases implicitly, as shown in Figure 9.

6 Circular implementation

> The representation and meaning levels are mutually-supporting. Representation provides the structures needed to describe meaning; meaning provides behaviour needed to implement the methods within the representation. The resulting dynamic, pervasively late-bound system achieves the goals of simplicity, openness, evolutionary adaptability, and user-centred implementation.
>
> End-user systems can be constructed by 'vertical extension': piling more layers into the end-user abstraction, joining the implementation chain at some appropriate point. Alternatively, 'horizontal mutation' uses the dynamic introspective capabilities of the basic system to modify its global behaviour from within, until it resembles the desired end-user system. Applying the second of these techniques locally (in a specific evaluation context) provides flexible syntax, semantics and pragmatics best adapted to expressing arbitrarily small parts of an application — as easily as as defining a function or macro in a traditional language.

A self-sustaining COLA architecture consists of two 'level's. The first deals with messaging between objects. As explained earlier, these objects describe their own behaviour; the semantics of message passing (and any pervasive object morphology necessary to support it) are defined by patterns of messages exchanged between objects. (This level says nothing, however, about the implementation of methods.) The essential purpose of this level is to represent symbolic expressions (syntactic structures) as illustrated in the upper portion of Figure 10.[19]

The second 'level' deals with the meaning of (behaviour implied by) these syntactic structures . Structures are repeatedly transformed into more fundamental forms (expressions, abstract instructions, intermediate forms, machine instructions), until an executable representation is reached. This is illustrated in the lower portion of Figure 10.

Completing the circular implementation follows directly from using the lower 'meaning' level to describe the internal behaviour of the method implementations (and the binding operation between object and message name) in the upper 'representation' level. This is illustrated in Figure 11.

COLAs are therefore entirely self-describing (i.e., self-implementing); from the representation of objects and the meaning of messaging, through parsing (a/message/lambda forms as well as any additional forms for convenience and connectivity, e.g., platform headers), to the compilation of syntactic forms into executable representation. Supporting a new language is achieved by finding a transformation connecting the input structures to those of the back end. This is illustrated in Figure 12.

The traditional (and obvious) approach appeals to 'upward vertical extension' of the implementation chain: adding one or more stages that explicitly transform the source form (text, bytecodes, ASTs) into the normal semantic structures of the host COLA for evaluation. The former will converge closely on the latter at some stage of the implementation chain, and this is equivalent to using the COLA as a high-level 'assembler' — albeit with much nicer properties than a target language such as C. Some core part of a basic COLA has no choice but to follow this model, as illustrated in the upper half of Figure 12. However, given a self-describing, dynamic, and pervasively late-bound host system implementation, a very different approach

[19] The objects provided by this level do not necessarily have any relationship with the objects of a particular end-user programming language implemented in the COLA, but are necessarily those used internally to express the syntactic constructs of the language, their syntactic, semantic and pragmatic interpretation, and ultimately their implementation.

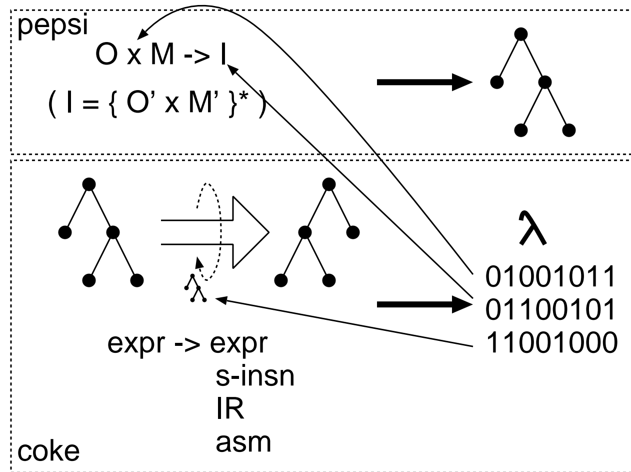

Figure 11: The behaviour of methods in the representational layer (including the messaging operation), as well as the (trivial) extrinsic mechanism that drives the transformation of semantic expressions (s-exprs) according to s-exprs, is implemented in s-exprs. This results in an entirely self-describing implementation.

---

is possible (and desirable).

The novel (and more subtle) approach mutates the COLA from within, causing it to assume the personality of the source language. The transformations applied by the COLA to its input structures are modified dynamically, *in-situ*, until the implementation chain (from top to bottom) corresponds to the natural structures and semantics of the input language.[20] This approach is illustrated in the lower half of Figure 12.

The second of these approaches in particular is scalable, and can be applied globally or locally (Section 5.1). Applying it locally provides scoped, domain-specific languages in which to express arbitrarily small parts of an application (these might be better called *mood-specific languages*). Implementing new syntax and semantics should be (and is) as simple as defining a new function or macro in a traditional language.

6.1 Bootstrapping

Bringing up a COLA is a four-phase process, illustrated in Figure 13. The first two phases yield a 'fake' (static only) implementation of the object model. Since this isn't The Real Thing, it's code-named 'Pepsi'. The second two phases bring up a completely dynamic system (code-named 'Coke') first statically (implemented in Pepsi) and then dynamically (implemented in Coke).

1. Bootstrapping the representation (object/messaging) layer. Since this assumes no previous implementation, it is performed in a foreign language. The current prototype's

[20]Those parts of the original system not required in the mutated system disappear in a puff of garbage collection.

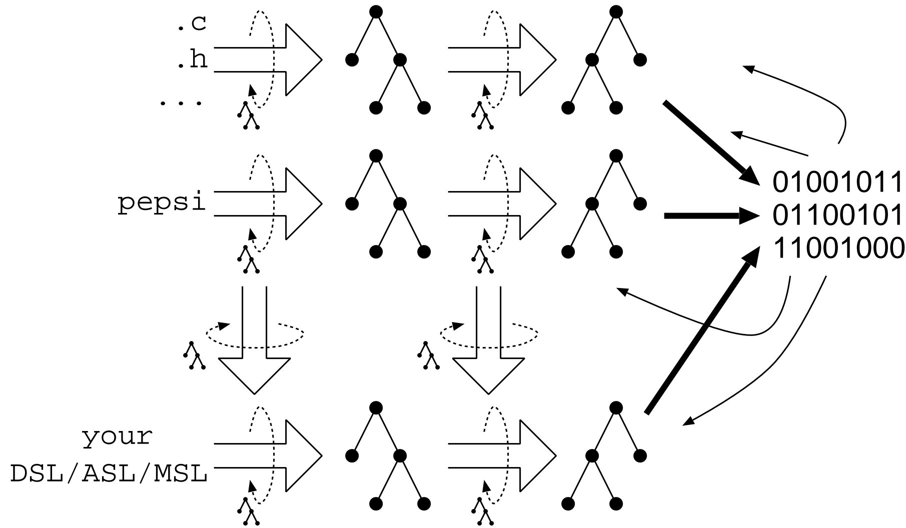

Figure 12: The principle of self-implementation is applied throughout the system, from interaction with the environment (parsing of platform and library header files) through the implementation of the representation and semantic expressions, to the final application. The ability to transform the running system from within itself leads to language/system implementations that mutate the initial system into the desired system (rather than being layered on top of it). The ability to transform dynamically the system at any level, either globally or locally, leads to solutions in which domain-, application-, or even mood-specific languages are constructed on-demand and on-the-fly, according to the needs of the system/application designer at any given moment.

'Phase One' consists of a complete Pepsi compiler written entirely in C++ (`Pepsi.C++`). The target language is C, used as a 'portable high-level assembly language'.

2. Phase Two is a set of kernel object types (`Object.pepsi`) and a complete Pepsi compiler (`Pepsi.pepsi`) written in Pepsi, using the Phase One (C++) compiler. (Once this compiler is producing byte-identical output to the Phase One compiler, the C++ version can be jettisoned without remorse.)

3. Pepsi-in-Pepsi is used to implement the dynamic semantics (behaviour-describing) layer. While this yields a working COLA capable of describing systems with dynamic semantics, its own implementation is still static (written in Pepsi). Therefore...

4. The semantics-describing parts of the Coke implementation (pretty much everything more interesting than the intrinsic behaviour of the structural representation layer) are reimplemented in Coke. The result is a self-describing, self-implementing, extremely late-bound (deeply dynamic) system cast in a form that can be easily mutated at any level into an implementation for a different language/system.

The resulting COLA is well adapted to the description and implementation of dynamic languages and systems in which end-user object models are either identical to, or orthogonal

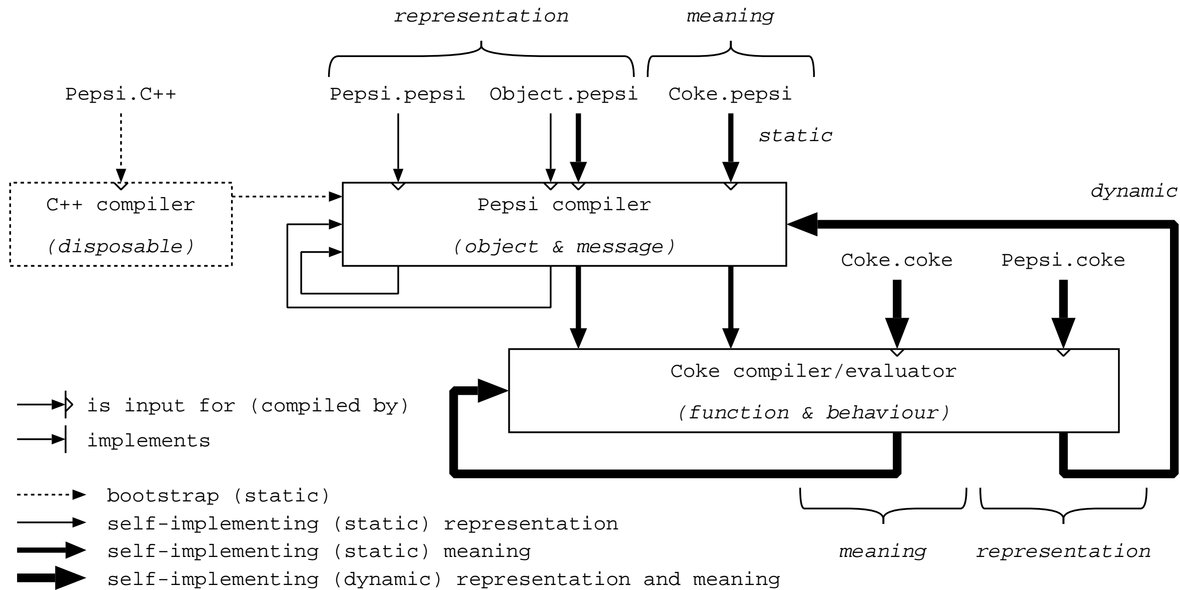

Figure 13: Bootstrapping the circular implementation. A disposable compiler written in C++ implements 'Pepsi' objects and messaging, and is immediately used to compile a Pepsi compiler written in Pepsi. Syntactic and semantic representations, along with mechanisms for their dynamic transformation, are implemented in Pepsi. The resulting system can then be applied to the implementation of Pepsi itself, yielding 'Coke' (The Real Thing). All aspects of the system (from dynamic binding in the method cache and intrinsic message lookup through to the most abstract transformations of the semantic representation) are now visible to, and dynamically modifiable by, the user.

---

to (but not extensions of) the intrinsic 'Pepsi' object model. For maximum flexibility, including the ability to specialise the intrinsic Pepsi object model to adapt it to arbitrary end-user applications, it is beneficial at this point to reimplement the methods of the Pepsi object implementation using Coke, freeing the Pepsi implementation entirely from any requirement that it be limited to Pepsi syntax/semantics. At this point, every component in the architecture and implementation is late-bound (can be modified, locally or globally, at runtime).

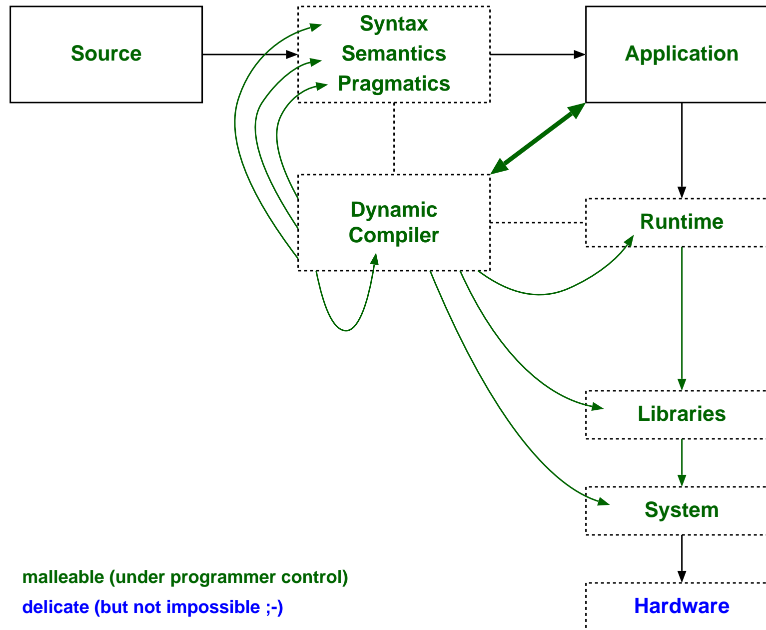

Figure 14: Unconventional programming languages. In green: a single, homogeneous, dynamic, late-bound implementation gives the user has control over all phases of application development, deployment and execution. Compatible dynamic services can be injected into the kernel, although modifying the hardware remains tricky (but not impossible, if flexible hardware such as FPGAs are available and have corresponding COLA back ends).

---

7 Unconventional programming languages

> Within a COLA, a single representation and pervasive, dynamic, late-bound paradigm gives users control over all aspects of implementation and execution. There are no barriers to expression or creativity. On the other hand, not having the 'stable' (and impenetrable) base of a more traditional system imposes additional responsibilities on the user. A useful compromise is to create traditional language/system implementations within a COLA, to give users some stability while preserving their freedom to modify any and all aspects of the language/system, at any depth, at any time.

Figure 14 illustrates a COLA-based programming language, in which the user has control over all phases of implementation. A single representation and paradigm controls program transformation (source to executable) and the runtime system that supports it, as well as their implementation and that of end-user code running within the system. Nothing is static, no aspect of the system is early-bound or rigidly defined/implemented, and nothing is (necessarily) hidden from (inaccessible to) the user. The system implementation and runtime are first-class components of the running application — or to look at it another way, the entire application is just an extension of its own implementation mechanism.

Compatibility with the C ABI ensures seamless integration with, and control of, the system environment. Dynamic compilation services, at any level of abstraction, can be encapsulated

and loaded into the kernel to provide runtime extension and modification of the operating system itself. Novel architectures (FPGA-based, for example), combined with an appropriate COLA 'code' generator, could have their hardware (or the division of labour between hardware- and software-implemented algorithms) reconfigured dynamically at runtime, by re-interpreting (unmodified) semantic structures within a different (or differently-parameterised) context.

The user's programming environment is homogeneous: there are no artificial distinctions between language implementation, runtime and application and no artificial barriers to expression or creativity.

A potential complication of this unconventional approach is confusion and disorientation on the part of conventional users (more accustomed to rigid, 'black box' systems). There are no *a priori* fixed points of reference; everything is (extreme) homogeneity, generality, flexibility and freedom.

Programming with COLAs must therefore commence by artificially creating points of reference — adding syntactic and semantic sugar (appropriate to the application domain) over the homogeneous base to form a stable base on which to build end-user systems.[21] There is no reason to preclude existing, more traditional languages and systems from such 'stable bases', however with the inestimable advantage that expedient (or maybe opportunistic) modification of the normally 'concrete' aspects of the language/system, at any depth in the implementation hierarchy, remains entirely possible.

[21] Such a base already exists in the basic system, in the languages and abstractions of the representation and meaning levels. Conversely, the entire system is organised to encourage the user to begin by tearing them down and replacing them with something more appropriate to their application domain.

8 Conclusion

Conventional languages and systems are incompatible with 'open future' principles. Built-in, early-bound assumptions are pervasive and form an impenetrable barrier to evolution (innovation, expression, integration). VM- or interpreter-based systems are little better; whereas the compiler is often available to the user, the interpreter rigidly defines both bytecodes (or some equivalent executable representation) thus preventing semantic exploration, and primitives thereby limiting pragmatic extensions.

An alternative is to create a virtual execution environment implemented entirely in (one of) the dynamic, late-bound language(s) that it implements. The kernel implements the simplest possible dynamic language, defining the minimum required of objects to support messaging. This single abstraction is pervasive, from application to metal, and being late-bound it puts the implementation chain entirely in the user domain. Respecting (unless overridden) the local C ABI puts the standard libraries and system services under direct control of the user.

Back ends that support the generation of static (offline compilation) and dynamic (incremental compilation) code encourage VM- or interpreter-based systems to be built without a VM or interpreter. Static compilation of a number of 'kernel' classes or types replaces the traditional monolithic VM or interpreter. A single execution model means static code exhibits dynamic behaviour, giving VM-like benefits (incremental programming, reactivity) equally for applications (user-level code) and runtime (the pre-compiled 'kernel' objects), while eliminating barriers to exploration, integration, and evolution.

Previously disenfranchised users are thus empowered to engage in wonderfully unreasonable behaviour, due to the potential for intercession at any level of their system's implementation. Expression is not limited by (and can encompass to any degree required) environmental, semantic, or pragmatic details, and no bottlenecks remain between the user and full exploitation of platform resources and services.

8.1 Omissions

Not (yet) covered in this document:

- a compelling example or two up-front, e.g: transforms implementing message send (combination of compile-time and run-time logic) with inline caches (compile-time allocation) in a procedural language, from message send through to generated code;
- selectors as compile-time and run-time objects (e.g., for implementing dispatch on multiple arguments);
- the type system, that eliminates the need for 'primitive' methods in the object system and provides additional scope for clever dispatch techniques (e.g., compile-time static overloading);
- (appendix with) details of the base semantic transforms (definitions, operators, conditionals, sequences, lambda, temporaries, scopes), the abstract machine (VPU), and the assemblers.

8.2 Status

Pepsi prototyped (without Coke layer on top, for now) with limited backends (C-based assembler of static code, but with correct dynamic behaviour), little debugging support, pending

platform integration (C99 parser for headers written but not integrated), no structure interpreter or bytecode support, and limited memory management. Coke prototyped (with C++ object system, not Pepsi, below) with dynamic only (no static code generation) for several stock CPU architectures. To the above extent what is written here is fact; anything else is desire and speculation, and the most effective way to find out if it's possible (or even desirable) is just to try to do it.

Lots of work remaining: integration of all of the above (with reimplementation as required), plenty of peripheral support (user interface, persistence), and some decent end-user theory of sharing and composition (traits, for example). Not to mention building something real (Croquet, for example, along with the various scripting languages we'd like to see supported in it) on top of it all.

8.3 Perspectives

A mature end-user theory of composition is required; traits would seem to be a very good candidate. Several end-user languages and systems should be implemented to demonstrate the viability, efficiency, and advantages of COLAs. Generic facilities for runtime feedback and dynamic optimisation are needed. Dynamic scanning and parsing algorithms (in the same spirit as the tree compiler transformations) are highly desirable. Alternative IRs, especially those better suited at extracting parallel behaviour for multiprocessors and distributed computations, would be interesting.

Community involvement would be an excellent forcing function for much of the above. It would also give some indication as to the problems of programmer education and user acceptance posed by the genericity, malleability and openness of COLA-based platforms, because...

> "Liberty means responsibility. That is why most men dread it."
>
> — George Bernard Shaw, *Maxims for Revolutionists*