LIL: CLOS reaches higher-order, sheds identity and has a transformative experience

François-René Rideau
Google
tunes@google.com

ABSTRACT

LIL, the Lisp Interface Library, is a data structure library based on Interface-Passing Style. This programming style was designed to allow for parametric polymorphism (abstracting over types, classes, functions, data) as well as ad hoc polymorphism (incremental development with inheritance and mixins). It consists in isolating algorithmic information into first-class interfaces, explicitly passed around as arguments dispatched upon by generic functions. As compared to traditional objects, these interfaces typically lack identity and state, while they manipulate data structures without intrinsic behavior. This style makes it just as easy to use pure functional persistent data structures without identity or state as to use stateful imperative ephemeral data structures. Judicious Lisp macros allow developers to avoid boilerplate and to abstract away interface objects to expose classic-looking Lisp APIs. Using only a very simple linear type system to model the side-effects of methods, it is even possible to transform pure interfaces into stateful interfaces or the other way around, or to transform a stateful interface into a traditional object-oriented API.

1. INTRODUCTION

In dynamically typed languages such as Common Lisp or Python (but also in some statically typed languages like the initial C++), programmers usually rely on ad hoc polymorphism to provide a uniform interface to multiple situations: a given function can accept arguments of many types, then dispatch on the type of these arguments to select an appropriate behavior. Object-oriented programming via user-defined classes or prototypes may then provide extension mechanisms by which new types of objects may be specified that fit existing interfaces; this extension can be incremental through the use of inheritance in a class (or prototype) hierarchy. More advanced object systems such as the Common Lisp Object System (CLOS) have further mechanisms such as multiple inheritance, multiple dispatch, and method combinations, that allow for a more decentralized specification of behavior.

In statically typed languages such as ML or Haskell (but also in some dynamically typed languages such as in PLT Scheme when using units (Felleisen 1998)), programmers usually rely on parametric polymorphism to write generic algorithms applicable to a large range of situations: algorithmic units can be parameterized with types, functions and other similar algorithmic units. These units can then be composed, allowing for elegant designs that make it easier to reason about programs in modular ways; the composition also enables the bootstrapping of more elaborate implementations of a given interface type from simpler ones.

In the past, many languages, usually statically typed languages (C++, OCaml, Haskell, Java, Scala, etc.), but also dynamically typed languages (PLT Scheme (Flatt 1998)), have offered some combination of both ad hoc polymorphism and parametric polymorphism, with a variety of results. In this paper, we present LIL, the Lisp Interface Library (Rideau 2012), which brings parametric polymorphism to Common Lisp in a way that nicely fits into the language and its existing ad hoc polymorphism, taking full advantage of the advanced features of CLOS.

In section 2, we describe the Interface-Passing Style (Rideau 2010) in which LIL is written: meta-data about the current algorithm is encapsulated in a first-class interface object, and this object is then explicitly passed around in computations that may require specialization based on it. We show basic mechanisms by which this makes it possible to express both ad hoc and parametric polymorphism.

In section 3, we demonstrate how we use this style to implement a library of classic data structures, both pure (persistent) and stateful (ephemeral). We show how our library makes good use of Interface-Passing Style to build up interesting data structures: ad hoc polymorphism allows us to share code fragments through mixins; various tree implementations can thus share most of their code yet differ where it matters; parametric polymorphism allows the composition of data structures and the bootstrapping of more efficient ones from simpler but less efficient variants; first-class interfaces allow the very same object to implement a given type of interface in different ways.

In section 4, we show how adequate macros can bridge the gap between different programming styles: between syntactically implicit or explicit interfaces, between pure functional and stateful data structures, between interface-passing and object-oriented style. All these macros allow programmers to choose a programming style that best fits the problem at hand and their own tastes, while still enjoying the full benefits of Interface-Passing Style libraries. They work based on a model of the effects of interface functions according to a simple type system rooted in linear logic.

We conclude by describing how Interface-Passing Style in Lisp relates to idioms in other programming languages and compares to existing or potential mechanisms for polymorphism in these languages or to their underlying implementation, and what are the current limitations of our library and our plans of future developments.

ILC 2012 October 26–27, Kyoto, Japan.

2. INTERFACE-PASSING STYLE

2.1 Using Interfaces

2.1.1 *Interface Passing: An Extra Argument*

For the user of a library written in Interface-Passing Style, interfaces are just one extra argument (more rarely, two or more) passed as the first argument (or arguments) to appropriate function calls. Each such interface argument provides these functions with some contextual information about which specific variant of some algorithm or data structure is being used.

As a syntactic convention followed by our library, symbols that denote interface classes, variables bound to interface objects, or functions returning interface objects will usually start and end with respective angle brackets `<` and `>`. For instance, the interface to objects that may be empty is `<emptyable>`, whereas a prototypical interface variable would be `<i>`.

2.1.2 *Trivial Example: Maps*

The most developed API in our library currently deals with (finite) maps, i.e. a finite set of mappings from keys to values. Our examples will mainly draw from this API. Maps notably include traditional Lisp alists (association lists) and hash-tables.

Thus, whereas a traditional object-oriented API might feature a function

```
(lookup map key)
```

that would dispatch on the class of the object `map` to determine how said map associates a value to the given `key`, an interface-passing API will instead feature a function

```
(lookup <i> map key)
```

where information on which precise algorithm to use is instead encapsulated in the extra argument `<i>`, an interface.

You could thus look up the year in an alist of data about a conference with code such as:

```
(lookup <alist>
  '((name . "ILC") (year . 2010) (topic . "Lisp"))
  'year)
```

In this case, the library-exported variable `<alist>` is bound to an interface for association lists.

Similarly, to insert a new key-value mapping in an existing map, you would call a function

```
(insert <i> map key value)
```

and depending on what interface `<i>` you specified, the generic function (in the sense of CLOS) would select an algorithm and appropriately update the data structure bound to variable `map`, returning the expected results.

2.1.3 *Pure vs Stateful Interfaces*

Of course, which effects a function call has and which results it returns may vary with the interface as well as the function.

For instance, our simplest map interface, `<alist>`, as the name implies, implements maps as association lists in the usual Common Lisp tradition: a list of pairs (`cons` cells), each specifying a key (as the cell's `car`) and a value (as the cell's `cdr`). Our alist interface is pure, meaning that the maps it manipulates are never modified in place, but new lists and association pairs are created as required. In particular, the function `insert` when applied to our `<alist>` interface, will return a new alist object:

```
(insert <alist>
  '((name . "ILC") (year . 2010) (topic . "Lisp"))
  'year 2012)
```

will return

```
((name . "ILC") (year . 2012) (topic . "Lisp"))
```

or some equivalent alist, without modifying any previous data cell, instead reusing the unmodified cells where possible.

If instead of alists, we had been using the interface `<hash-table>` and a hash-table object, the function `insert` would have returned no values, instead modifying the existing hash-table in place.

Because `insert` means something quite different for pure and stateful data structures, with incompatible invariants, our library actually defines two different generic functions, `pure:insert` and `stateful:insert`, each in its own package. [1]

By contrast, there is only one function `interface:lookup` that is shared by all pure and stateful interfaces and imported in both the `pure` and `stateful` packages. Indeed, lookup has the same specification in both cases: it takes an interface, a map and a key as parameters, and it returns two values, the value associated to the given key if a mapping was found, and a boolean that is true if and only if a mapping was found.

It is easy to underestimate the importance of the semantic differences between pure and stateful data structures, until you've actually tried to gloss over them. See in section 3 a more detailed justification why we keep such a clear distinction between the two.

2.1.4 *First-Class Interfaces*

Interfaces are first-class objects. They don't have to be compile-time constants. Functions can abstract over interface objects, create new interface objects, etc. By abstracting over the interface object, accepting it as an argument and passing it to other functions, you can write algorithms that are independent of the specifics of the underlying data structure.

For instance, you can write functions that fold over a map (reduce it) without knowing how the map is implemented internally (or indeed whether it's pure or stateful, computing serially or in parallel); and you can apply such functions to a wide variety of situations, in each of which the map's implementation may be suitably optimized to the context at hand:

```
(defmethod sum-values ((<i> pure:<map>) map)
  (let ((submaps (divide/list <i> map)))
    ;; see promised invariant of divide/list
    (cond
      ((null submaps) ; no element
       0)
      ((null (rest submaps))
       ;; only one mapping, extract its value
       (nth-value 1 (first-key-value <i> map)))
      (t ;; general case: recurse and map-reduce
       (reduce #'+
         (mapcar (λ (m) (sum-values <i> m))
                 submaps))))))
```

The method above abstracts over interface `<i>`, which is constrained to be a sub-interface of `pure:<map>`, and will notably rely on the latter's signature function `divide/list`. This function is defined in our library; it divides a map `map` into a list of non-empty submaps each with strictly fewer mappings than the original

[1] Packages are the standard first-class namespace mechanism of Common Lisp. The syntax for a symbol can either leave the package implicit, or explicitly specify a package name as a prefix followed by one or two colons and the symbol name.

map, unless said map has exactly one mapping, in which case it returns a singleton list containing that map. The above method could be trivially parallelized by replacing `mapcar` and/or `reduce` by parallelizing, queueing variants.

2.1.5 *Functions of Multiple Interfaces*

Most algorithms are parameterized by a single interface object. Indeed, when multiple interface objects are required, they are usually uncurried into a single argument, with reader methods allowing to extract the former interfaces from the argument and its parameter slots. For instance, instead of tree-manipulating functions requiring two interface arguments, one to describe the tree structure and another one to describe the key order, such functions will typically require only one interface argument, encapsulating both tree structure and key order; then, from a given interface object `<my-tree>`, the key interface can typically be obtained by evaluating `(key-interface <my-tree>)`.

Still, some functions naturally require several interfaces as arguments. This typically happens when writing algorithms that bridge between two different domains, rather than when dealing inside a given domain. For instance, the function `convert` is defined as follows:

```
(defgeneric convert
    (<destination> <origin> object)
  (:documentation "Convert an OBJECT
  following interface <ORIGIN> into a new object
  following interface <DESTINATION>."))
```

Thus, an algorithm working on finite maps will typically take a single interface argument, and assume all maps used inside the algorithm use the same representation strategy described by this interface; but an algorithm that specifically federates access to several databases will typically take several interface arguments, one per federated database... and will typically return a single interface (and possibly an object that follows this interface) so that normal algorithms can access it all through that single interface (and object, if any).

2.1.6 *Caveat: No Type Checking*

Interface-Passing Style in Common Lisp has a definite low-level feeling, in that the user is given both full control and full responsibility with respect to passing around appropriate interfaces, which is compounded the fact that Common Lisp has dynamic typing rather than static typing, so that the evaluator will not issue errors or emit warnings at compile-time if you pass interfaces and arguments that do not match each other.

The downside is that if the user fails to ensure consistency between the interfaces being used and data structures being passed as arguments, unspecified behavior may ensue (usually resulting in a runtime error at some point), as generic functions may or may not check their arguments for consistency. While our library does provide a function `check-invariant` as part of the signature of interface `<type>`, most of the methods we provide do not call said function, which in general is rather expensive (with a cost increasing at least linearly with the size of the object), and instead trust users to call it as appropriate, typically at the entry points of their code, and often only while testing or debugging.

The upside of this lack of automatic type-based interface control is that the user can explicitly specify an interface in some uncommon cases where the "canonical" interface that could have been deduced by type inference isn't what he wants, and even in cases where there isn't any such "canonical" interface to begin with.

2.2 Defining Interfaces

2.2.1 *define-interface*

Interfaces can be defined with the `define-interface` macro, which is an extension to `defclass` that handles several features specific to interfaces.

For instance, here is a stripped-down excerpt from our library:

```
(define-interface <emptyable> (<type>) ()
  (:abstract)
  (:generic empty (<emptyable>)
   (:values object) (:out 0)
   (:documentation "Return an empty object"))
  (:generic empty-p (<emptyable> object)
   (:in 1) (:values boolean)
   (:documentation "Is object empty?")))
```

It defines an interface `<emptyable>`, the name of which is passed as first argument to the macro, as in `defclass`.

The second argument is a list specifying super-interfaces from which to inherit behavior, also as in `defclass`. In this case, there is one and only one super-interface, `<type>`. In our library, `<type>` is an abstract interface specifying that some datatype is targeted by interface functions. `<emptyable>` extends it with the notion that elements of that type may be empty.

Still as in `defclass`, the third argument is a list of slots. A non-empty list of slots is how parametric polymorphism is achieved. In this case, this list is empty, as there are no parameters defined by this interface.

Finally, and there again like `defclass`, `define-interface` accepts a list of options. In addition to the regular `defclass` options, it also recognizes a few of its own. For instance, the `<emptyable>` interface above uses the `:abstract` option, to declare that it doesn't implement all its declared functions and must not be instantiated.

This interface also uses the `:generic` option to declare two generic functions that are part of the signature of the interface, `empty` and `empty-p`. Furthermore, for each function, a return value convention may be defined as well as a calling convention. Indeed they are defined in this case: the first function takes no argument beyond the interface; the `(:values object)` specifies that it returns exactly one value, named object, and the `(:out 0)` specifies that return argument in first position is of the target type (indexes are 0-based). Similarly, the second function is a predicate, takes one argument of the target type and returns one boolean value.

We will now go over each of these features in more detail.

2.2.2 *Inheritance of Interfaces*

Interfaces may inherit from other interfaces and be organized in inheritance hierarchies.

For instance, `<type>` is an interface with an associated datatype. `<eq>` is an interface that inherits from `<type>`, for datatypes with an equality comparison predicate `==`. `<hashable>` is an interface that inherits from `<eq>`, for datatypes with a function `hash` such that two equal values (as compared by `==`) have the same hash. `<equal>` is an interface that inherits from `<hashable>`, and implements equality with the standard Common Lisp predicate `equal` and `hash` with the standard Common Lisp function `sxhash`. `<eql>` is an interface that inherits from `<eq>`, and implements equality with the standard Common Lisp predicate `eql`; since there is no standard Common Lisp hash function that corresponds to it, it doesn't inherit from `<hashable>`.

2.2.3 *Multiple Inheritance of Interfaces*

Interfaces may inherit from any number of super-interfaces. Indeed, our interfaces are CLOS classes, and since CLOS supports multiple-inheritance, they may easily inherit from multiple other such classes. Interfaces may only inherit from other interfaces. Internally they are instances of the CLOS metaclass `interface-class`.

As an example of multiple inheritance, our `pure:<tree>` map interface inherits from both `interface:<tree>`, an interface specifying read-only signature functions on trees, and `pure:<map>`, an interface specifying signature functions for maps with pure update as well as mere lookup.

2.2.4 *Interface Mixins*

Our library also relies on multiple-inheritance extensively in the form of mixins, also known as traits in other programming languages: small interface classes implement a small aspect of the interface. Oftentimes, a mixin will be used to simply deduce the implementation of some signature functions from other signature functions. Depending on which signature functions are more "primitive" for a given concrete data structure, converse mixins may be used that deduce some functions from the others or the other way around.

For instance, the `<eq>` interface actually has two associated functions, `(== <i> x y)` that compares two objects `x` and `y`, and equivalently `(eq-function <i>)` that returns a function object that may be passed as an argument to various higher-order functions. The mixin `<eq-from-==>` will automatically deduce `eq-function` from == while the converse deduction is provided by the mixin `<eq-from-eq-function>`.

2.2.5 *Parametric Interfaces*

Interfaces may be parameterized by other interfaces as well as by any object.

For instance, consider the current definition of `<alist>` in our library:

```
(define-interface <alist>
    (<map-empty-as-nil>
     <map-decons-from-first-key-value-drop>
     <map-update-key-from-lookup-insert-drop>
     <map-divide/list-from-divide>
     <map-map/2-from-fold-left-lookup-insert-drop>
     <map-join-from-fold-left-insert>
     <map-join/list-from-join>
     <map>)
  ((key-interface :type <eq>
    :initarg :key-interface
    :reader key-interface))
  (:parametric (&optional (eq <eql>))
     (make-interface :key-interface eq))
  (:singleton))
```

The super-interface list contains several mixins to deduce various methods from more primitive methods, together with the interface `<map>` that provides the signature.

But most importantly, the list of slots contains a single slot `key-interface`. Indeed, association lists crucially depend on an equality predicate with which to compare keys when looking up a given key. Our `<alist>` interface therefore has this slot, the value of which must be an instance of a concrete sub-interface of `<eq>`, that will specify how to compare keys.

Slot definitions such as these are how we achieve parametric polymorphism in Interface-Passing Style: an interface class with such a slot get instantiated as interface objects with a specific value in that slot, and a method defined on this interface class can extract the value in said slot as a parameter to its behavior.

Our definition of `<alist>` also uses two options recognized by `define-interface` that are not provided by `defclass`: `:parametric` and `:singleton`.

2.2.6 *Concrete Parametric Interfaces*

The `:parametric` option automatically generates a function to instantiate parameterized interface objects. This function further uses memoization so interfaces with identical parameters end up being the same interface object rather than a new object every time.[2]

In the above `<alist>` example, the function takes one optional parameter that defaults to `<eql>` (itself a variable bound to a singleton interface). This means that if no `<eq>` interface is specified, we will follow the Common Lisp convention and tradition in providing the `eql` function as the default comparison function. The body of the parametric function creates the interface object using the locally defined function `make-interface` that handles memoization of an underlying CLOS `make-instance`.

Our library implements data structures more elaborate than alists. For instance, you could use a balanced binary tree, in which case you would have to provide the tree interface with a parameter `key-interface` that inherits from `<order>`, so that keys may be compared. Thus, `(stateful:<avl-tree> <number>)` will return an interface that is ideal to maintain a sorted index of numbered records, whereas `(pure:<avl-tree> <string>)` is suitable to build persistent dictionary structures. However, if you want your dictionary not in ASCIIbetical order but rather in a proper collating sequence for Japanese, you'll have to build an interface `<japanese-collation>` around a Unicode library, and pass it as an argument to `pure:<avl-tree>`, or to a variant thereof that caches the collation key.

2.2.7 *Singleton Interface Variable*

The `define-interface` extension option `:singleton` automatically defines a special variable bound to a canonical instance of a concrete interface class. If a `:parametric` option was provided, its function will be called with default values. Otherwise, a trivial version of such a function will be defined and used.

Clients can therefore use the variable `<alist>` to refer to the one default such interface, instead of having either to create a new instance every time, be it with `(<alist>)` or using `(make-instance '<alist> :key-interface <eq>)`.

2.2.8 *Multiple Dispatch*

Because CLOS has multiple dispatch, our generic functions can dispatch on more than the first argument, thus preserving the language's object-oriented style on arguments beyond the initial interface argument. In a language with single-dispatch, we couldn't do that, at least not directly, as dispatching on the interface would use up the object-oriented ability to specialize behavior depending on arguments.

As the simplest example, an interface `<empty-object>` could implement the `<emptyable>` signature functions as follows, given a class `empty-object` for its empty objects:

```
(defmethod empty-p
    ((<i> <empty-object>) (x t))
  nil)
```

[2] This memoization is effectively a hash-consing strategy. It works because interfaces don't usually have intensional identity, only extensional content. Indeed, they embody behavioral meta-information notionally meant to be expanded before any code is actually run. See in section 4.3 how interfaces compare to traditional objects.

```
(defmethod empty-p
    ((<i> <empty-object>) (x empty-object))
  t)
```

Non-empty objects would be matched by the first method, while empty objects would be matched by the more specific second method.

More complex examples could involve more methods, with bigger class hierarchies or dispatch on more than two arguments.

2.2.9 *Interface Signatures*

Each interface is attached to a set of functions declared as part of the interface's signature. The functions from the interface's super-interfaces are inherited; additional functions can be directly declared using the `:generic` option of `declare-interface`.

Some interfaces, such as `<emptyable>` above, exist for the sole purpose of declaring such functions, while leaving full freedom to sub-interfaces as to how to implement them. That is why `<emptyable>` was marked as `:abstract`: it is an error to try to instantiate it, its purpose is to be inherited from by other interfaces, and dispatched upon in some methods.

We saw that some abstract interfaces have the opposite purpose: they implement one or several signature functions in terms of other signature functions, that may be more "primitive" in various concrete interfaces.

Finally, some interfaces do implement the complete declared signature, either directly or through inheritance of appropriate mixins. They are concrete interfaces meant to be instantiated, such as `pure:<alist>` above. Instantiation usually happens through a function declared by the `:parametric` option or a variable declared by the `:singleton` option. These options are mutually exclusive with the `:abstract` option.

The fact that some interfaces are concrete is one notable difference between our interfaces and interfaces in other languages such as Java. Another notable difference is that interfaces are not to be implemented by a class of objects, with the first argument to every function in the signature being treated specially and having to be of an object class implementing the interface. Instead, our signature functions treat all arguments uniformly, and none of these arguments have to be restricted to any particular class.

In particular, with our approach of detached interfaces, there is no problem whatsoever with having "binary methods"; no special status is required for "constructor" methods that create an object of some target type when there was no object yet on which to dispatch methods; and there is no dilemma regarding contravariance or covariance of types when inheriting from a parametric interface. Interface-Passing Style solves these issues by making subtyping between interfaces independent from subtyping between data structures, eschewing the need to resolve the sometimes contradictory constraints between the two kinds of subtyping. Note that many of these issues could be avoided or glossed over in Common Lisp thanks to its multimethods and dynamic typing; however, our approach could solve these issues even in a language without single dispatch and/or with static typing; and indeed, an essentially equivalent approach already solves these issues in Haskell.

3. REVISITING CLASSIC STRUCTURES

3.1 Pure and Stateful Data Structures

3.1.1 *Pure, Stateful, their Intersection, and Beyond*

We built LIL, the Lisp Interface Library, with the ambition that it should become the definitive library for data structures in Common Lisp. While we initially chose Interface-Passing Style to achieve parametric polymorphism, which was not previously available in Common Lisp, this style was also helpful to address other issues in developing our library.

For instance, so that we may improve on all existing libraries, we decided to provide both pure (functional) (persistent) data structures and stateful (imperative) (ephemeral) data structures. Furthermore, we decided to do the Right Thing™ by sharing as much as possible of the interface and implementation between these two styles of data structures, with APIs congruent enough with each other that it is possible to build automated bridges between the two styles.

The interfaces in Interface-Passing Style proved to be a great locus at which to formalize both the commonalities and divergences between pure and stateful data structures.

3.1.2 *Common Interfaces: Read-Only Access*

The `interface::<map>` interface directly declares functions `lookup`, `first-key-value`, `fold-left`, `fold-right` and `map-alist` that access an existing map in a read-only way (the latter builds an alist from the map). It also declares a function `alist-map` that creates a map initialized from an alist, and is quite useful for specifying non-empty constant maps. These functions are applicable to pure as well as to stateful maps. There are also inherited functions such as `check-invariant`, `empty` and `empty-p`.

Thus, it is possible to write generic read-only tests for map data structures that work for all map implementations, pure as well as stateful. Indeed, LIL includes such tests in its test suite.

3.1.3 *Interface Divergence: Update*

We mentioned how the two distinct functions `pure:insert` and `stateful:insert` have different signatures as far as return values go. The same difference exists between the `pure:drop` and `stateful:drop` functions: both have the same input signature

```
(<map> map key) (:in 1).
```

But whereas the former has the output signature

```
(:values map value foundp) (:out 0),
```

the latter has the output signature

```
(:values value foundp) (:out t).
```

This means that the pure function returns an updated version of the original map data structure as its first return value, whereas the stateful function omits this return value and instead side-effects the map passed as input argument.

Other functions that update data structures have similar differences in their signatures: pure methods tend to return new updated data structures as additional values, whereas such return values are omitted by stateful methods that instead update existing data structures in place through side-effects.

3.1.4 *Keeping Pure and Stateful Apart*

It was a deliberate decision to avoid further unification between the pure and stateful interfaces, and to not make them follow the exact same convention for return values as well as for calling arguments. Indeed our very first API had fewer divergences than it now does; however, after a lot of experimentation, we discovered many convergent reasons why it is a good idea to maintain a very clear separation between the two:[3]

- Most important of all, publishing interfaces that have identical signatures yet essential semantic differences (i.e. side-

[3] It was suggested we name our two packages `church` and `state` rather than `pure` and `stateful`, to insist on the need to keep them separate.

effects versus no side-effects) is an invitation to confused, erroneous programs: functions will be written that look like they work in both cases and get invoked as if they did, yet somewhere along the way they will make crucial assumptions about the presence or absence of side-effects, and they will fail when called with the wrong kind of interface.

- The only programs that would work in *both* cases are programs that strictly follow the functional paradigm while following the linear logic discipline that no object is ever modified more than once nor read after it has been modified. Our API still allows users to write such programs, using the pure interface; and thanks to the transformers we describe in section 4 these programs may use or provide interfaces to stateful data structures.

- Therefore, maintaining a fake compatibility of calling convention between these two APIs with actually different semantics is often detrimental and never useful. It must be avoided. That's a case where punning causes confusion without bringing expressive power.

- Moreover, it is more consistent both with previous practice of stateful OO APIs and with the principle of least redundancy that no value should be returned that is specified to always be identical to an input argument, where for stateful methods, identical usually means `eq`. (Notable exceptions in our API are `divide` where the initial map is returned as second value, or `divide/list` where it is returned (if not empty) as the first element of the result list; these exceptions are in the sake of preserving another invariant, that the results of division can be respectively be `join`'ed or `join/list`'ed back into the original map.)

- Abiding by these simple principles allowed the transformations in section 4, between pure and stateful interfaces and between Interface-Passing Style and object-oriented style, to work based on a simpler effect language than might otherwise have been needed. In this case, simpler is better not only because it makes things easier to explain in this article, but also because it already took a month to write and debug the initial 175-line macros with 19 nested levels of binding forms that lie at the heart of these transformations. [4]

- Making for clearly different interfaces between pure and stateful makes for a better demonstration of the adapter between pure and stateful interfaces. Our transformers would still work and be required if the interfaces were the same, but the confusion between the two similar-looking interfaces might make it harder to explain what is going on versus what isn't, while hiding the fact that our transformations work even if the interfaces differ.

3.1.5 *Incremental Layers of Functionality*

We have striven to implement our data structures in small incremental layers by taking full advantage of CLOS features such as multiple inheritance, multiple dispatch and method combinations.

For instance, here is an abstract mixin that adds a layer of self-balancing to a binary tree, taking advantage of CLOS `:after` method combination:

```
(define-interface <post-self-balanced-binary-tree>
    (<binary-tree>) ()
  (:abstract))

(defmethod insert :after
    ((i <post-self-balanced-binary-tree>)
     node key value)
  (declare (ignore key value))
  (balance-node i node))

(defmethod drop :after
    ((i <post-self-balanced-binary-tree>)
     node key)
  (declare (ignore key))
  (balance-node i node))
```

And here is how stateful AVL trees are implemented on top of previous layers:

```
(define-interface <avl-tree>
    (interface::<avl-tree>
     <heighted-binary-tree>
     <post-self-balanced-binary-tree>) ()
  (:abstract))

(defclass avl-tree-node
    (interface::avl-tree-node
     heighted-binary-tree-node) ())

(defmethod node-class ((i <avl-tree>))
  'avl-tree-node)

(defmethod balance-node ((i <avl-tree>)
                         (node empty-object))
  (values))

(defmethod balance-node
    ((i <avl-tree>) (node avl-tree-node))
  (ecase (node-balance node)
    ((-1 0 1) ;; already balanced
     (update-height node))
    ((-2)
     (ecase (node-balance (left node))
       ((-1 0))
       ((1)
        (rotate-node-left (left node))))
     (rotate-node-right node))
    ((2)
     (ecase (node-balance (right node))
       ((-1)
        (rotate-node-right (right node)))
       ((0 1)))
     (rotate-node-left node))))
```

The superclasses already handle the read-only aspect of AVL trees (mostly invariant checking, in this case), and the stateful aspects of maintaining the tree height and having to rebalance after updates. The only incremental code we need is the specification of how to rebalance nodes.

This decomposition of interfaces into lots of incremental interfaces makes for a very clean programming style where each interface is a small mixin that is quite easy to understand.

Note however the current burden of having to explicitly maintain two class hierarchies, one for the interfaces, and one for each type of object that the interfaces may manipulate. We have hopes of eliminating this boilerplate in the future, by having `define-interface` manage for each interface such a set of object classes; but we have not started work on such a solution yet.

3.2 Interface Tricks and Puns

A clear disadvantage of Interface-Passing Style, as compared to means by which other languages achieve similar expression, is that

[4] Such deep nesting is the best case we've seen for the use of `nest`:

```
(defmacro nest (&rest r)
  (reduce (λ (o i) `(,@o ,i)) r :from-end t))
```

Note with our system `lambda-reader` one can actually use λ instead of `lambda`.

it imposes upon the user the cost of keeping track of the interface objects that are passed around. But as a trade-off, there are some advantages to balance the equation. We are going to enumerate a few of these advantages, from the most trivial to the most advanced.

3.2.1 Dispatch without Object

As a first advantage, interfaces as separate detached arguments allow users to manipulate data where there is no object to dispatch on. This is very clear in the case of the `pure:<alist>` interface: it operates on primitive entities (`cons` cells, `nil`) without requiring these entities to be full-fledged objects, and without requiring the user to retroactively add a superclass to these entities for dispatch purposes. (Note however that normal CLOS multiple dispatch also works without this limitation.) This is also clear for constructor functions, where no object exists yet on which to dispatch.

3.2.2 Bootstrapping Data Structures

A second advantage of explicit interfaces is that different instances of a given interface class can apply to the same object. One way that we put this technique to profit on LIL is in how we bootstrap pure hash-tables.

A hash-table is a generic implementation of a finite map with fast access time, supposing the existence of a hopefully fast hash function (typically mapping keys to integers) as well as an equality predicate. The hash function will hopefully distinguish with high probability any two unequal objects that may be used as keys in the map. The location of the entry mapping the key to its value can then be quickly computed from the key hash; in case several keys collide (have the same hash), some compensation strategy is used, such as putting all those keys in the same bucket, or using an alternate key.

In the well-known stateful case, the key-indexed table is typically implemented as a random-access array with $O(1)$ access time. In the pure case, the key-indexed table will typically be a balanced binary tree, which has slightly worse $O(\log n)$ access time but allows for persistent data structures (i.e. old copies are still valid after update).

The Common Lisp standard specifies a class `hash-table`, but this only provides a stateful variant of hash-tables. We built an interface `stateful:<hash-table>` that matches the signature of `stateful:<map>` while using those standard hash-tables underneath, but also needed a `pure:<hash-table>` as a generic pure map mechanism.

Our `pure:<hash-table>` is constructed in a straightforward way from the principles we recalled above: from a slow but generic map interface mapping keys to values (generic meaning that keys can be anything) and a fast but specialized map interface mapping key hashes to key buckets (specialized meaning that keys are integers), we bootstrap a fast generic map interface mapping of keys to values; we achieve this by composing the above together with the fast implementation handling the common case and the slow implementation handling the collisions. By default our slow generic map implementation is `(<alist> <equal>)` and our fast specialized map implementation is `(<avl-tree> <number>)`, under the nickname `<number-map>`; but these parameters are under user control.

Our `pure:<hash-table>` is defined parametrically as follows (the following paragraphs are to be read in package `pure`):

```
(define-interface <hash-table>
    (<map-join-from-fold-left-insert>
     <map-join/list-from-join>
     <map-update-key-from-lookup-insert-drop>
     <map-map/2-from-fold-left-lookup-insert-drop>
     <map>)
  ((key-interface :type <hashable>
    :reader key-interface
    :initarg :key)
   (hashmap-interface :type <map>
    :reader hashmap-interface
    :initarg :hashmap)
   (bucketmap-interface :type <map>
    :reader bucketmap-interface
    :initarg :bucketmap))
  (:parametric
       (&key (key <equal>)
             (hashmap <number-map>)
             (bucketmap (<alist> key)))
    (make-interface :key key
      :hashmap hashmap :bucketmap bucketmap))
  (:singleton)
  (:documentation "pure hash table"))
```

Methods are then straightforward. For instance, see the `insert` method, and notice the pun:

```
(defmethod insert
    ((<i> <hash-table>) map key value)
  (let ((hash (hash (key-interface <i>) key)))
    (insert
     (hashmap-interface <i>) map hash
     (insert
      (bucketmap-interface <i>)
      (multiple-value-bind (bucket foundp)
          (lookup (hashmap-interface <i>)
                  map hash)
        (if foundp
            bucket
            (empty (bucketmap-interface <i>))))
      key
      value))))
```

Indeed the very same object `map` is passed as an argument to the same function `insert` through two different `<map>` interfaces: the outer one is the original `<i>` which is a `<hash-table>`; the inner map is `(hashmap-interface <i>)`, which is presumably a `<number-map>`. There is a third call to `insert`, with the interface `(bucketmap-interface <i>)` which is presumably a `<alist>`; however, this time its argument is not `map` but the proper hash bucket, which was a value in the `map` object seen with the inner interface, or a new empty bucket if none was found.

Note, however, that even though the punning is nice, and can potentially save both in memory and in API complexity, systems that disallow punning might allow us to express the same concepts via an indirection. For instance, in Haskell, you could hide the "same" underlying structures under different unary constructors to distinguish the various stages of such a bootstrap. The important property of parametric polymorphism is that *interfaces are compositional*. You can build new interfaces by composing existing interfaces, abstracting away patterns and reproduce them automatically; you are not limited to agglutinating exponentially ever more unrelatable cases.

In the near future, we would like to bootstrap more data structures this way, for instance following some of the algorithms documented by Chris Okasaki (Okasaki 1996).

3.2.3 Same Data, Multiple Interfaces

As a more general kind of pun, an object can be in the target type of several interfaces, not necessarily instances of the same interface class. There is no need to wrap and unwrap data inside constructors of different classes to see that data through a different viewpoint;

simply change the interface, and the same data can be punned into meaning usefully different things. For instance, one can see an array as a sequence of elements one way, or the reverse way; one can view it as the support for a binary queue or a hash-table. No need to shuffle around the array elements in memory, or to indirect access to the array through several view or façade objects that have to be managed and still may cause extra allocation. Just look through the lens of a different interface.

We haven't yet implemented any non-trivial example of such heavy punning. The following two paragraphs are simply ideas we have for future work.

Because an interface is not tied to the data, the data can remain unchanged while the interface changes. This way, some algorithms can be simplified by factoring data access through a single more compact data structure visited with a finite or evolving set of interfaces. In extreme cases, punning data with multiple interfaces can make a program work simply where naive wrappers would fail by leaking memory, and where non-leaky wrappers would require additional complexity through layers of "optimization" or memoization in such wrappers.

Punning data with multiple interfaces could also help build composite data structures, whereby each node in the object graph is part of several structures: a hash-table for fast retrieval by key, a priority heap for scheduling a job queue, a sequence for a consistent enumeration order, some lazily balanced tree for retrieval with a more rarely used key, etc. We suspect that to make such non-trivial punning easy, we will have to build new infrastructure, to manage the heavily-punned classes that implement such composite structures incrementally yet without extra boxing. This will involve having a model of class labels associated to an interface, and a list of mixins attached to each of these labels; when instantiating a concrete interface, the system will have to ensure that an actual class exists for each of these labels, that inherits from each of these mixins and no more; for extra punning, there may be the need to rename some of the mixins to avoid clashes.

Obviously, anything that can be done through the formal use of interfaces can be done without, in the first come Turing tar-pit of a programming language. Still we contend that interfaces can be an elegant solution to many problems, particularly in situations where a problem has symmetries and regularities that can be expressed as the same parametric interface applying to multiple situations.

4. INTERFACE TRANSFORMATIONS

4.1 Making Interfaces Implicit or Explicit

4.1.1 Making Interfaces Implicit in a Scope

Even though arbitrary first-class interface objects can be passed as argument in any function call, it is quite often obvious from the context that an interface object under consideration is going to be passed around in each and every call to a function in that interface's signature, or almost every such call.

Therefore, we provide a macro evaluating a body of code in a lexical scope in which it is syntactically implicit that a given object `interface` is being passed around in all calls to some functions specified by `functions-spec`:

```
(with-interface
  (interface functions-spec &key prefix package)
  &body body)
```

The `functions-spec` is a compile-time constant that can be a list of function names, but more often than not is the name of single interface class, in which case it denotes all the names of the functions declared as part of the signature of that interface class. Other keyword arguments provide control over which package is to be used for fast aliases (by default the current one, for shorter names) and what optional prefix to use (by default none as that would defeat the purpose).

For instance, the following code implements an insertion sort for number-indexed alists, by inserting all its entries in an ordered tree then walking the tree entries in order:

```
(defun mysort (alist)
  (with-interface (<number-map> <map>)
    (let ((m (alist-map alist)))
      (fold-right m #'acons nil))))
```

Notice how the `<number-map>` interface object was implicitly passed to calls to two functions in the `<map>` interface class, `alist-map` and `fold-right`. This macro of course gets more interesting as you write longer functions that have more such calls. (Interestingly, since the repeated insertion is hidden behind the generic function `alist-map`, this function works in both pure and stateful contexts.)

4.1.2 Implicit Interface in Method Definition

Many interfaces have methods implementing their declared functions in the context of which this situation definitely applies: the interface argument will be passed unchanged to other methods in the interface signature. Therefore `define-interface` also has an option `:method>` that defines methods with an implicit `with-interface`. For instance, here is the definition of the previously mentioned `<eq-from-==>` mixin:

```
(define-interface <eq-from-==> (<eq>) ()
  (:abstract)
  (:method> eq-function ()
    (λ (x y) (== x y))))
```

Notice how the interface argument is omitted from the lambda-list. Notice also how no interface argument is explicitly passed to `==`.

In case the interface is needed for some explicit call, the interface argument is bound to the symbol naming the interface (in this case `<eq-from-==>`), rather than to a special symbol (such as `self` as in other languages). In case one of the shadowed interface functions is needed for some call with an explicit interface different from the implicit one, the symbol naming this function can be called (in this case `eq-function`), since in Common Lisp its global binding isn't shadowed. Obviously, the adaptation of this syntactic facility to a language different from Common Lisp would require a different solution, such as requiring use of a name prefix when invoking the long-form functions.

4.1.3 Global Elision of Interface Argument

Once you have built an interface that is perfect for a lot of your algorithms, instead of passing it around over and over, you can make it altogether globally implicit. Choose a package and/or a prefix, and use the macro:

```
(define-interface-specialized-functions
  interface functions-spec &key prefix package)
```

It will create in the current or specified package some global functions (with optional prefix added to their name) that internally call the specified interface functions, implicitly passing around your global interface object.

For instance, if you find yourself using pure hash-tables a whole lot, you could create a package `pure-hash-table` in which you would evaluate:

```
(define-interface-specialized-functions
  pure:<hash-table> pure:<map>)
```

and voilà, all the functions you need are there for you to use in that package.

4.1.4 Making Interfaces Explicit

It might happen that you have some classes that implement in a classic object-oriented style some interface that you are interested in using as a parameter. Then you may have to define a singleton interface with wrapper methods adapting between the two APIs.

If it happened that the APIs were indeed identical but for the extra argument, a macro could be trivially written to automatically provide for the adaptation. However, in practice, the case doesn't happen, because odds are low a legacy or third-party object-oriented interface will exactly match your modern Interface-Passing Style signature. And odds are similarly low that if you're interested in the flexibility of interfaces, you would start with the more rigid object-oriented style and need to convert to the more flexible Interface-Passing Style, rather than start with Interface-Passing Style and extract an object-oriented API from there through one of the above or below mechanisms.

4.2 From Pure to Stateful and Back

In previous sections, we explained how interfaces maintain meta-information about the call arguments and return value conventions of functions in their signature. We also saw that the signature of the pure variant of an interface was systematically related to the signature of the stateful variant of the "same" interface. What if we could formalize this systematic relation? Then this meta-information would be more than mere documentation: we could implement automatic correspondences between the pure and stateful variants of an interface.

This is what we have implemented in LIL: we have built a model of what effects declared interface functions have on objects of the targeted interface type. Within the constraints of this model, we can automatically emit wrappers that convert between pure and stateful interfaces.

4.2.1 Mutating and Linearized

In a pure (functional) interface implementing a persistent data structure, input arguments are values that are never modified. Instead, some functions have output values that represent an updated value for the "same" notional object as one of the input values. In a stateful (imperative) interface to an ephemeral data structure, input arguments are objects that may be inspected read-only or modified in-place; functions that update an object modify it in place and do not usually return a new object.

The correspondences between these two styles are as follows. From a pure interface, a stateful interface may be deduced by putting the persistent values in a mutating box that stores the current value of the object; given a box, a value is extracted from the box into the input, and an update value if any is put back into the box on output. We call the above transformation mutating and its result the *mutating* interface. From a stateful interface, a pure interface may be deduced by putting ephemeral values in a linearized box that ensures any value is only modified once, and not used thereafter; the object is extracted from the box into the input, and is invalidated if there are any modifications, while a fresh box is created to hold the object in its new state if modified. We call the above transformation linearize and its result or argument (depending on context) the *linearized* interface.[5]

Interestingly, a stateful data structure linearized then mutating is isomorphic to the original data structure; however, a pure data structure mutating then linearized isn't isomorphic to the original, unless we require that users should make an explicit copy of the data structure each time it may be used more than once, as per Linear Logic. Indeed, the mutating transform is all about introducing the discipline of an object having a single current value that is only used once to produce the new current value (unless explicitly copied), and the linearized transform is all about enforcing the discipline that any value may only be used once (unless explicitly copied). Now, the entire point of (pure) persistent data structures is usually that they make copying a data structure practically free, and that using a data structure multiple times is made free by copying it implicitly as needed; therefore this limitation in how the two transforms aren't quite inverse of each other is as designed.

4.2.2 Trivially Modeling Effects

LIL has a very simple model of the effects that a function may have, the simplest with which we could get results:

- Some input arguments and output values are marked as being of the interface-targeted type.
- Each input argument is put in correspondence with either an output value or `nil` or `t`.
- An output value can be in correspondence with one input argument only; it can be in correspondence with none or equivalently with `nil`.
- A correspondence between input argument and output value means that the output has the same identity as the input after possible modifications.
- A correspondence between an input argument and `nil` means that the argument may be read but not modified.
- A correspondence between an input argument and `t` means that the argument may be modified.
- A correspondence between an output value and `nil` means that the value is created.

Syntactically, the marking happens in the `:generic` declaration of `define-interface`. A `:in` keyword introduces a list of input arguments or `nil` markers. A `:out` keyword introduces a list of output values or `nil` or `t` markers. The correspondence is simply that the nth element in one list corresponds to the nth element in the other, or `nil` if the other list is shorter.

Keeping things really simple, this model only considers effects on required arguments; our model cannot express effects on `&optional` arguments, `&rest` arguments, `&key`word arguments. [6].

This model is as simple as can be, and yet it fits most of the functions in our map API.

[5] The pure functions can be seen as the state-passing style expansion of implementing the imperative interface with an explicit state monad. Stateful functions can be seen as pure linear functions with some arguments and results made implicit. The transformations are all about making these details implicit or explicit, depending on which way you go.

[6] In Common Lisp, the list specifying how arguments are bound to what variables when a function is invoked is called a lambda-list. A lambda-list may specify required arguments, then optional arguments introduced by `&optional`, then a rest argument introduced by `&rest`, then keyword arguments introduced by `&key`. We remember the lambda-list of the input arguments the function accepts, and we record a lambda-list of the output values it returns, which may be considered as the lambda-list of the function's continuation.

4.2.3 *Pure Interface in a Mutating Box*

LIL includes a macro to automatically transform a pure interface into a stateful interface. For instance, here is how we define a mutating map interface, parameterized by the pure map interface that implements its underlying operations:

```
(define-mutating-interface
  <mutating-map> (stateful:<map>) (pure:<map>)
  ()
  ...
  (:parametric (interface)
    (make-interface :pure-interface interface)))
```

The first argument is the name of the new interface. The second argument is a list of super-interfaces of the new stateful interface being created by mutating. The third argument is a list of super-interfaces of the underlying pure interfaces being wrapped. The fourth argument is a list of slot definitions and overrides for parametrization, completing what's implicit in mutating. What remains is a list of options to `define-interface`; elided are several manual method definitions for functions that our macro fails to automatically wrap; included is a `:parametric` function definition.

The macro defines a new interface class, and wrapper methods for all declared interface functions with a matching name between the pure and stateful packages that also have declared effects as per our model.

Values are put into an object box containing the current value. As seen in the examples below, we use a function `box!` that takes one argument and creates a box object with a mutable slot `value` initialized with the argument; the slot can be read with `box-value` which takes the box as argument and returns its value; they can be written with `set-box-value` which takes the value and the box as arguments and sets the box value.

The wrapping of a read-only function works by extracting the value from the box and passing it to the pure function. For instance, the cleaned up[7] macroexpansion of the wrapping for `lookup` is as follows:

```
(defmethod lookup
    ((<interface> <mutating-map>) map key)
  (let* ((<pure-interface>
            (pure-interface <interface>))
          (pure-map (box-value map)))
    (multiple-value-bind (value foundp)
        (lookup <pure-interface>
                pure-map key)
      (values value foundp))))
```

When a function updates an old value into a new one, we simply extract the updated value from the pure function's results and store it into the box. For instance, the cleaned up wrapper for `insert` is:

```
(defmethod stateful:insert
    ((<interface> <mutating-map>) map key value)
  (let* ((<pure-interface>
            (pure-interface <interface>))
          (pure-map (box-value map)))
    (multiple-value-bind (updated-map)
        (pure:insert <pure-interface>
                     pure-map key value)
      (set-box-value updated-map map)
      (values))))
```

Finally, if a new object is created, we grab the value returned by the pure function and put it in a box, such as in this wrapper for `empty`:

```
(defmethod stateful:empty
    ((<interface> <mutating-map>))
  (let* ((<pure-interface>
            (pure-interface <interface>)))
    (multiple-value-bind (pure-empty)
        (pure:empty <pure-interface>)
      (let* ((empty-object (box! pure-empty)))
        empty-object))))
```

Not only is this transformation useful, it is how our stateful alists are implemented: indeed, the naive direct implementation of alists without boxing falls short when you want to add entries to an empty list, for whereas non-empty alists are `cons` cells with state and identity, the empty list is represented as `nil` which has neither. Boxing is the correct way to do stateful alists: it has essentially the same performance profile, doesn't require ugly hacks to specially handle empty alists, and with our transformer, it minimizes the need to write redundant code.

4.2.4 *Manual Method Transformation*

Unhappily, our very simple model for effects cannot cover methods with more advanced calling conventions. Our transformation macros allow for users to manually specify methods where our automation falls short or fails.

Interestingly, amongst the many functions we initially came up with while developing our map API, the only that didn't fit this simplest of models were `join/list` and `divide/list`. The former respectively takes a list of map objects as argument, and the latter returns a list of map objects.

Here is a how we manually wrap `divide/list`:

```
(:method> stateful:divide/list (map)
   (let ((list
          (pure:divide/list
            (pure-interface <mutating-map>)
            (box-value map))))
     (when list
       (set-box-value (first list) map)
       (cons map
             (mapcar #'box! (rest list))))))
```

Note how the first element in the list is special in that it shares the identity of the map being divided, which is part of the contract of `divide/list`. (LIL, following Common Lisp tradition, neither imposes nor provides any means to automate the enforcement of these contracts.)

Also note that as a limitation in our current transformation macros, methods in the original and transformed APIs are simply matched by name. In the future, it would be easy to allow the user to customize the way method names are processed, transformed or overridden during these transformations.

[7] The clean up we did is for readability only. The actual macroexpansion uses gensyms; instead we renamed gensyms and other symbols so they are more explanatory. The actual macroexpansion has `(declare (ignore ...))` clauses; we omit such clauses when no variable was ignored. The macroexpansion also includes trivial renaming of variables to bridge between the calling conventions of the inner and outer functions; we beta-expand these renamings away, and omit binding forms without non-trivial bindings. In presence of rest or keyword arguments, the macroexpansion uses `apply` for the inner function and/or for `values`; it uses `funcall` in absence of such arguments; we simplify the `funcall` case into a direct call, and do away with unnecessary `values` statements. Finally, we omit some the package of symbols where it isn't relevant to our explanation. But we neither simplify a `multiple-value-bind` with a single binding into a `let` nor merge it with a previous or subsequent binding forms, as it would blur rather than demonstrate the general pattern of the macro.

4.2.5 *Stateful Interface in a Linear Box*

The reverse transformation works in a very similar way. For instance, stateful map interfaces are transformed into linearized pure map interfaces as follows:

```
(define-linearized-interface
  <linearized-map> (pure:<map>) (stateful:<map>)
  ()
  (:method> join/list (list) ...)
  (:method> divide/list (map)
     (let ((list
            (stateful:divide/list
             (stateful-interface <linearized-map>)
             (box-ref map))))
       (and list
            (mapcar 'one-use-value-box list))))
  (:parametric (interface)
    (make-interface
     :stateful-interface interface)))
```

Everything works in a way similar to the mutating transformation. We elide the body of the `join/list` manual method, but offer the `divide/list` manual method for contrast with the reverse transformation.

Note that `one-use-value-box` is a one-argument function that creates a box with a slot value initialized to that argument, that can be read many times with `box-value`, but is used up and not further readable when read by `box-ref`. We use the latter invalidating read function before any operation that modifies the contents of the box; therefore, it is invalid to try to access an old version of the wrapped object. If you want to keep a version of an object for future use, you must explicit copy its contents into a new object before you make any modification, as per Linear Logic.

Here are the cleaned up macroexpansions for the wrappers around `lookup`, `insert` and `empty` respectively:

```
(defmethod lookup
    ((<interface> <linearized-map>) map key)
  (let* ((<stateful-interface>
          (stateful-interface <interface>))
         (stateful-map (box-value map)))
    (multiple-value-bind (value foundp)
        (lookup <stateful-interface>
                stateful-map key)
      (values value foundp))))

(defmethod pure:insert
    ((<interface> <linearized-map>) map key value)
  (let* ((<stateful-interface>
          (stateful-interface <interface>))
         (stateful-map (box-ref map)))
    (stateful:insert <stateful-interface>
                     stateful-map key value)
    (let* ((updated-map
            (one-use-value-box stateful-map)))
      updated-map)))

(defmethod empty
    ((<interface> <linearized-map>))
  (let* ((<stateful-interface>
          (stateful-interface <interface>)))
    (multiple-value-bind (empty-object)
        (empty <stateful-interface>)
      (let* ((one-use-empty
              (one-use-value-box empty-object)))
        one-use-empty))))
```

4.2.6 *Using Transformed Maps*

Mutating or linearized interfaces are not just a mathematical curiosity, they have applications to actual systems.

For instance, a stateful algorithm may sometime involve snapshotting the state of objects; if the objects are big or if snapshotting happens often enough, the usual stateful data structures can be prohibitively expensive; but by simply wrapping a purely functional persistent data structure designed to make copying essentially free, you can remove such a speed or space bottleneck. And all you need to do is to start using a mutating interface instead of the vanilla stateful interface. Using Interface-Passing Style, you can also easily defer this kind of decision until you know enough about the constraints of your application, and revise the decision after these constraints evolve.

Conversely, you may have great algorithms developed in a functional style that allow them to combine easily and to apply to situations beyond the limitations of linear state. Yet, some of these algorithms may also apply within the limitations of linear state, in which case you may want to use them together with the less cumbersome stateful programming style. A linearized interface allows you to use your functional library with your stateful data structures.

4.2.7 *Limits of Our Effect Model*

Our effect model is sufficient to cover a complete API for the manipulation of maps, both pure or stateful. Indeed, the `divide/list` and `join/list` functions, which it did not cover, can be considered as convenience optimizations for what can be done without, using fixed-arity functions `divide` and `join`. Still, we have already reached the limits of our model, and we must mention how our model may be fixed to handle such cases.

Our signature annotations can be seen as some very simple first-order linear type system. We believe that our automatic transformations can be formalized as functors, and that it is possible to generalize both our model and our transformations as part of some higher-order type system rooted in Linear Logic. Efforts toward such a generalization would probably be an interesting venue for further research, but are beyond the scope of our current projects.

4.3 From Interfaces to Classes and Back

4.3.1 *Interfaces as Detached Classes*

An object-oriented API is a set of classes and generic functions operating on objects, objects having at the same time identity, data content, and behavior attached to them; behavior of generic functions happen by dispatching on the class of the first object (and sometimes those of subsequent objects). An Interface-Passing API is a set of interfaces, datatypes and generic functions operating on data that may or may not have identity; behavior is attached to interfaces, and generic functions dispatch primarily on the first interface (and sometimes subsequent interfaces).

One way of looking at things is by distinguishing concerns of behavior (code and meta-data) and state (data and identity). Interface-Passing Style separates them, with the interface carrying only the behavior. Object-oriented Style conflates them, with an object carrying all of it. A correspondence can be drawn between Interface-Passing Style and traditional object-oriented Style by viewing an interface as "detached" class information, as the part of an object that doesn't include its state, and by viewing an object as a "subjective" interface, one where some state has been moved into the interface.

Using this viewpoint, it is possible to mechanically derive an Interface-Passing API from an object-oriented API or an object-oriented API from an Interface-Passing API. To go from one style to the other is a matter of splitting or joining back the behavior, identity and data aspects that the respective other style preferred to join or split. Depending on whether joining or splitting makes more

sense for a given API, it can be written in the style that yields the cleanest code, yet used by clients using the other style if needed.

An interface can be seen as an object that doesn't carry any identity or data but only class-related behavioral information. Any data slot of interface objects is then seen as a class parameter (as in a C++ template parameters), and dynamically created interface objects are akin to dynamically created first-class classes. Explicitly passing the interface around is as if an object's "virtual method table" or equivalent were passed as a separate argument. This is somewhat similar related to the *self* argument of many object systems, except than an interface includes all the meta-level class information but none of the identity and runtime data associated with the object. And it applies even when there is no self object (yet) with the identity or data to dispatch on (e.g. for constructor methods).

Conversely, you can view traditional objects as "subjective" interfaces, where no explicit state object is passed, but rather where any state has been moved inside the interface itself.

To extract an Interface-Passing API from an object-oriented API is easy: it suffices to introduce a dummy interface object, which can be done as per the above section 4.1.1.

To extract an object-oriented API from an Interface-Passing API is harder, but we can reuse the same effect system we developed above for that purpose: our objects will be boxes that at the same time have their identity, an attached data value, and a reference to the interface that is being transformed to object-oriented style. Function dispatch happens by locating the first object, extracting the interface, applying the corresponding interface function to the unboxed data, and wrapping new objects into boxes as appropriate.

4.3.2 Parametric Classification

LIL includes a macro to automatically transform a stateful interface into an object-oriented API, a process we dub *classification*. However, this macro requires a little bit of configuration by the user to deal with constructor methods for which there isn't an object to dispatch on.

Let us first examine the case where we want to generate a general-purpose object-oriented API out of a general-purpose abstract interface. For instance, here is how we export our `stateful:<map>` interface parametrically into a `>map<` class API, evaluating this in package `classified`:

```
(define-classified-interface-class
  >map< (object-box) stateful:<map>
  ((interface :initarg :interface))
  (:interface-argument (<interface> stateful:<map>)))
```

The wrappers for `lookup`, `insert` are then as follows:

```
(defmethod lookup ((map >map<) key)
  (let* ((<interface> (class-interface map))
         (map-data (box-ref map)))
    (multiple-value-bind (value foundp)
        (interface:lookup <interface> map key)
      (values value foundp))))

(defmethod insert ((map >map<) key value)
  (let* ((<interface> (class-interface map))
         (map-data (box-ref map)))
    (stateful:insert <interface> map key value)
    (values)))
```

So far, so good: in good object-oriented style, the behavior is controlled by the first object supplied, from which the interface was extracted. However, the wrapper for `empty` is awkwardly different:

```
(defmethod empty ((<interface> stateful:<map>))
  (multiple-value-bind (empty-data)
      (interface:empty <interface>)
    (let* ((object (make-instance '>map<
                     :interface <interface>
                     :value empty-data)))
      object)))
```

This difference reflects a general problem that object-oriented style has with constructors. Because object-oriented style locates class dispatch information in the first object, it has nothing to dispatch on where there is no object yet. That is why object-oriented style has to treat constructors specially. In CLOS, objects are constructed by the `make-instance` function, which is special in that it takes as its first argument a class meta-object (or a class name, resolving to the former). Unhappily, this makes `make-instance` (and constructors in general in other languages) a part of the meta-object protocol rather than of the interface protocol. Interface-Passing Style is more uniform there, which enables us to do automatic transformations of interface protocols that just work with constructors without having to special-case them. Interface-Passing Style also allows for several constructors in an interface to a sum type, without having to go through the hoops of having a single multiplexing constructor or having to have a different interface for each object type in the sum. For instance, in Interface-Passing Style, both `empty` and `cons` would be regular members of the interface signature for lists or sequences.

Nevertheless, our transformation from Interface-Passing Style to object-oriented style has to do something about constructors. Since in this case we are transforming an abstract interface, objects need to carry a parameter for the actual concrete interface with which the object was created. We use the slot `interface` for that, and the above definition overrides its default definition so it may be initialized with keyword `:interface`. This keyword is also the default value of the `:interface-keyword` option (which we don't override), and that tells our transformer how to pass the interface in its call to `make-instance`. When creating the object, we need to supply this interface to constructor functions. The `:interface-argument` option in `define-classified-interface-class` tells constructors such as `empty` to accept an extra argument which will become the interface to be attached to the constructed object. How the interface is extracted from that argument could have been overridden with the `:extract-interface` option, but we rely on the default, which is to use it directly. All this customization was necessary to generate the `empty` wrapper above.

Also note how, in this example, we constrain the argument to be of type `stateful:<map>` before we construct a `>map<` object, so that other methods of `empty` could construct other kind of empty objects based on a different interface. Thus, the API we extract from our parametric classification is truly object-oriented, and can be extended, or shared with other classes beyond our transformed interfaces.

4.3.3 Singleton Classification

The user may opt to create an object-oriented API out of a singleton concrete interface. Then, constructor functions do not need an extra argument to be supplied the interface: the interface is a constant that is wired into the function. We can specify it with the `:extract-interface` option, though there is no interface argument from which to extract it. For instance, we could create an API for a singleton interface `stateful:<number-map>`, by evaluating the following form in its own package `classified-number-map`:

```
(define-classified-interface-class
  >nm< (object-box) stateful:<number-map>
  ((interface :initform stateful:<number-map>
              :allocation :class))
  (:interface-keyword nil)
  (:extract-interface stateful:<number-map>))
```

Here the `:allocation :class` means that the interface slot is the same for all objects of that class. Indeed, when classifying an interface API, instance-specific data of the interface becomes class-specific data of the class of the manipulated objects. The `:extract-interface` option tells us how to get the interface in constructor methods despite the absence of extra interface argument. The `:interface-keyword` option, being overridden to `nil` instead of the default `:interface`, tells us that we don't need to provide an interface argument to the internal `make-instance` constructor, since it is a class constant rather than an object-specific parameter. We could have further customized object wrapping and unwrapping with the `:wrap` and `:unwrap` options; they specify the prefix of a form to build the object from interface data or extract the interface data from the object respectively; they default respectively to `'(make-instance ',class)` and `'(box-ref)`, where `,class` will actually be the name of the class being defined by `define-classified-interface-class`.

With the definition above, the wrapper for the `empty` constructor will then be:

```
(defmethod empty ()
  (let* ((<interface> <number-map>))
    (multiple-value-bind (empty-data)
        (interface:empty <interface>)
      (let* ((object (make-instance '>nm<
                       :value empty-data)))
        object))))
```

With such transformations of singleton interfaces, it becomes possible to develop libraries using the power of parametric polymorphism, composing simple parametric interfaces into more elaborate ones, and yet expose the result as a traditional API so users do not even have to know that Interface-Passing Style was used internally.

Note however that unless you follow some protocol to parameterize your constructors as in our parametric classification above, your API will not be object-oriented but simply imperative. Indeed, if constructors do not take an extra parameter but instantiate a constant class and interface, this part of the API is not extensible and cannot be shared with other classes. A future version of our library might have an option to add a prefix or suffix to names of constructor methods, so normal methods can be part of an object-oriented API without a clash because of unsharable constructors.

5. CONCLUSION

5.1 Related Work

5.1.1 Many Well-Known Predecessors

Interface-Passing Style is a novel tool that has proven particularly effective for implementing a generic data structure library in Common Lisp. Indeed, Interface-Passing Style was developed specifically to fit both the shortcomings and the assets of Common Lisp. But the underlying ideas are hardly original; both the interface aspect and the passing-style aspect of Interface-Passing Style have many predecessors in the tradition of programming languages.

The runtime objects that we expose as explicit user-visible interfaces are typically how existing implementations of languages with parametric polymorphism have implicitly implemented this feature for decades, under the hood. For instance, that is how Haskell implements Type Classes (Jones 1993), PLT Scheme implements Units (Felleisen 1998), and ML implements functors: an extra interface argument is implicitly passed around to the lower-level functions implementing these various constructs, and this extra argument encapsulates the parameters to said constructs.

As for passing-style, people who study the semantics of computer programs have long practiced the principle of reifying some previously implicit aspect of their programs into new explicit objects that are passed around, as a means to formalize the meaning of their computations. Continuation Passing Style is a famous instance of this practice, as are all kinds of environment passing styles.

5.1.2 Interface-Passing Style Specificities

However, there are several ways in which our Interface-Passing Style differs from any of the above-mentioned systems; these ways, some of them innovative, are all related to our embracing the powers and limitations of Common Lisp in implementing parametric polymorphism:

- *We do not rely on static information*, either purely syntactic (via scoping as in PLT) or somewhat semantic (via type inference as in Haskell), to statically resolve interfaces, or otherwise hide them behind a language abstraction. Instead, we embrace the dynamic nature of Common Lisp and let interfaces be first-class objects that may be determined at runtime at any call site. This can be viewed either as a restriction on the capabilities of our technique, or as the absence of a restriction on its applicability.

- *Interface arguments are passed around explicitly rather than implicitly*. We embrace the opening up of what in other systems is an implementation detail. This gives our library a low-level flavor of control and responsibility; while the responsibility is indeed sometimes burdensome, we can take advantage of that control to access the same data structure through multiple interfaces.

- *Our interfaces can be parameterized by arbitrary first-class data*. The parameters are not constrained to be second-class entities to allow for termination of a type inference algorithm. This does raise the difficulty for authors of compilers to optimize our code, or for authors of proof systems to accommodate the complexity.

- *We make it easy for users to hide these interfaces in usual cases* thanks to Common Lisp macros, with facilities both syntactic (such as `with-interface`) and semantic (such as our macros to go from interfaces to classes). In common cases, we can therefore eschew the burden of explicitly passing around interface objects.

- *We support ad hoc polymorphism by explicitly dispatching on interface arguments*. These interfaces need not be uniform dictionaries (like the implicit arguments in the respective implementations of the above-mentioned systems), but can be objects of arbitrary user-defined classes, subject to the usual object-oriented dispatch techniques.

- *Our ad hoc polymorphism is scoped outside of parameters, not inside*. This lambda lifting of interface objects matters a lot for Common Lisp, because Common Lisp has neither first-class class combinators nor cheap portable anonymous

classes, but instead has a global public namespace that favors dynamic linking of new methods to existing generic functions and dynamic instantiation of new interface objects with runtime parameters. Note that this starkly contrasts with classes inside parameterized units, as done in the PLT unit article (Flatt 1998), where parameterized classes are statically linked and strictly scoped via an assemblage of units; though the PLT approach allows for dynamic instantiation of unit assemblages with runtime parameters, any such assemblage is semantically sealed and unextensible after instantiation. Once again, the Common Lisp approach has a lower-level feel overall.

- *We rely on multiple-dispatch to not sacrifice object-oriented style when using interface dispatch.* In a language with single-dispatch, dispatch on our explicit interface argument will use up the programmer's ability to rely on ad hoc polymorphism to express his algorithms. In LIL, we leverage the multi-method capabilities of CLOS to dispatch on our interface objects and still be able to dispatch on further arguments as per normal object-oriented style.

Our solution would fit any other dynamic language, especially if it also has multiple dispatch and/or syntax extension facilities.

Our base design could also fit a static language, but it would be extremely painful unless the static type system were expressive enough, at which point language designers probably already have solutions to the issues we address. At that point, our contribution would probably be limited to inspiring the development of Interface-Passing Style libraries as an alternative to traditional object-oriented style.

5.1.3 Innovation: Interface Transformations

Thanks notably to the syntax extensibility of Common Lisp, we could also achieve a few interesting features beside the addition of parametric polymorphism to Common Lisp:

- Our library provides both pure and stateful data structures that share a common interface for read-only methods.
- We provide macros to make interfaces implicit again in the usual cases.
- For the sake of the above and below, we associate generic functions to interfaces.
- Additionally, we annotate generic functions with trivial metadata about their side-effects.
- Based on such effect types, we implemented automated transformations bridging between pure (persistent) and corresponding stateful (ephemeral) data structure.
- Based on the same effect types, we implemented automated transformations bridging between Interface-Passing Style and traditional object-oriented style.

While the ideas behind these features will sound quite well understood by people familiar with programming language theory, we are not aware of any existing library in any previous programming language that could in theory or would in practice leverage those ideas.

We hope our success will generate more widespread interest in supporting multiple programming styles with automated library transformations.

5.2 Current Status and Future Work

5.2.1 Current Status

LIL at this point is already a usable data structure library that has contributed features not previously available to Common Lisp users: it offers an infrastructure for users to develop their own parametrically polymorphic data structures, or to easily extend existing ones; and it provides a generic map interface with pure and stateful variants, and implementations as balanced binary trees, hash-tables or Patricia trees. Some of these data structures were not previously available in free software libraries for Common Lisp.

Yet, in many ways, LIL is still in its early stages; at the current moment it is a usable proof of concept more so than a full-fledged library. It sports as few usable features as necessary to illustrate its concepts, and each of its features is as bare as possible while remaining functional. There are thus many axes for development, both in terms of actually provided algorithms and in terms of linguistic abstraction.

5.2.2 Obvious Potential Improvements

Obviously, more known data structures could be provided: Stacks, queues, double-queues, arrays, heaps, sets, multi-sets, both pure and stateful, could be fit in the existing framework. We notably intend to port the algorithms from Chris Okasaki's now classic book (Okasaki 1996) and other currently popular pure functional data structures; and of course matching interfaces to well-known stateful variants. Also, provisions for safe concurrent access to data structures could be introduced.

Just as obviously, our linguistic features could offer more bells and whistles: users could have more flexibility in mapping names, parameters and other aspects of their interfaces when translating between variants of algorithms, pure and stateful, interface-based and class-based, single-threaded or concurrent, etc. These transformations could be more mindful of interface and class hierarchies rather than operating on all the generic functions of one pair of APIs at a time. The packaging of the current features could be improved, with internals being refactored and exported. We could use Context-Oriented Programming techniques (Costanza 2008) to dynamically bind extra implicit arguments to function calls and re-expose an Interface-Passing Style API as a classic object-oriented style API. Our interfaces could integrate with `deftype` and `typep` via the same technique we used to implement the `list-of` library.

Finally, we could try to study the performance impact of our code, and improve it where it matters. For instance, SBCL is known to dynamically optimize method dispatch; we could make sure that it does a proper job with our data structures. In case compilers have trouble with code in Interface-Passing Style, we could develop some protocol for partial evaluation that will ensure proper inlining is done.

5.2.3 More Advanced Projects

Now, here are a few less-obvious ways in which we'd like to improve LIL.

Firstly, we'd like to explore how algorithms can be developed in terms of combining small individual features, each embodied in an interface mixin: controlling whether any given property is implemented as a slot or a user-defined method; controlling whether some data is indirectly accessible through a box or inlined in the current object; combining multiple data structures to achieve better access guarantees (i.e. records are both nodes of a hash-table for constant-time lookup and of a doubly-linked list for preserving insertion order), or implementing the same data structure twice with a different view (i.e. records are part of several trees that in-

dex several fields or computed expressions from fields). Mixins would be combined in the style of `cl-containers` (contributors 2005) and its macro `find-or-create-class`, which implements first-class class combination on top of Common Lisp's second-class class object-system but first-class reflection. Some protocol would manage the several classes of objects associated to an interface, and combine them all (or the relevant subset) with additional mixins when such mixins are specified; this would also be used for extracting "classified" APIs from Interface-Passing Style APIs. Possibly, we could also provide some way for abstract interfaces to provide default concrete implementations; thus, in simple cases one could obtain a full implementation just by specifying the high-level properties of the desired algorithm, yet in more complex cases, manual specialization would be possible.

Second, we'd like to explore how both pure and stateful variants of some algorithms can be extracted from a single specification: the specification would essentially combine a pure node-building algorithm with annotations on which object identities are to be preserved between original and new objects. The pure variant would just create new values and drop the identities. A stateful variant would clobber old identities using change-class on previous nodes. New variants could purely pass around or statefully side-effect an additional explicit store object. The main ambition though is that a single specification should simply make all the variants possible, so that each user may fine-tuning which variant makes sense for him while being able to share his algorithms with other users that need the "same" algorithm viewed from a totally different angle.

Third, and relatedly, we'd like to explore how to improve the so far trivial language by which we currently express "effects" of API functions. The current first-order specifications can probably be generalized into some kind of higher-order type system. Presumably, API transformations could be automatically extracted from expressions in that more elaborate effect specification language. Possibly, simple API implementations themselves could in some cases be automatically extracted from the API specification itself. If the specifications also include annotations about performance guarantees, this opens a venue for a more declarative approach to data structure development.

5.2.4 Why And Wherefore

The proximate trigger for what became this article was a study we made on how to introduce modularity in the overly monolithic code base we were working on; we started the library as a proof of concept of our proposal for introducing parametric polymorphism in Common Lisp. Interestingly, though, the idea of detaching behavioral meta-data about objects in an entity separate from their state data and passed as an extra argument dates from our very first dabbling in implementing an Object Oriented language; indeed, our dissatisfaction with how traditional object-oriented style conflates behavior and state in the same "object" package-deal dates from the same time, as we were trying to figure out semantics for object systems and ways to modularly express mathematical concepts.

As for the goal we are aiming for, it is the automated unification of different programming styles: programmers shall be able to write incremental contributions each in a style most suited to express its meaning, yet be able to combine them all despite their being written in different styles. The program fragments would be automatically aligned along a common semantic framework thanks to declarative specifications of the style in which they are intended (some more constrained bits of code can be viewed in many ways). And it should thereafter be possible to seamlessly combine these contributions into a common result, made available to the user according to whichever point of view best suits his needs.

Bibliography

Gary King and contributors. cl-containers. 2005. `http://common-lisp.net/project/cl-containers/`

Pascal Costanza. Context-Oriented Programming in ContextL. 2008. `http://www.p-cos.net/documents/contextl-soa.pdf`

Matthew Flatt and Matthias Felleisen. Units: Cool Modules for HOT Languages. In *Proc. PLDI 98*, 1998. `http://www.ccs.neu.edu/scheme/pubs/pldi98-ff.ps.gz`

Robert Bruce Findler and Matthew Flatt. Modular Object-Oriented Programming with Units and Mixins. In *Proc. ICFP 98*, 1998. `http://www.ccs.neu.edu/scheme/pubs/icfp98-ff.pdf`

John Peterson and Mark Jones. Implementing Type Classes. 1993. `http://web.cecs.pdx.edu/~mpj/pubs/pldi93.html`

Chris Okasaki. Purely Functional Data Structures. 1996. `http://www.cs.cmu.edu/~rwh/theses/okasaki.pdf`

François-René Rideau. Interface-passing style. 2010. `http://fare.livejournal.com/155094.html`

François-René Rideau. lisp-interface-library. 2012. `http://github.com/fare/lisp-interface-library/`

Credits

I wish to thank my wife Rebecca for supporting me throughout this development, my employer Google and my manager Allan Fraser for bearing with me during this digression from my main tasks, Eli Barzilay for the Racket Scribble system, Jon Rafkind for giving me a template to start from, Eric O'Connor for kickstarting the development of LIL as an independent library, Zach Beane for being a one-man Release and QA system for Common Lisp libraries, my colleagues Arthur Gleckler, Scott McKay and Alejandro Sedeño, as well as Philipp Marek, for their careful proofreading, my anonymous reviewers and my other proofreaders for their feedback, and Kuroda Hisao for organizing the conference and pushing me to give my very best on this article.