Modular Module Systems: a survey

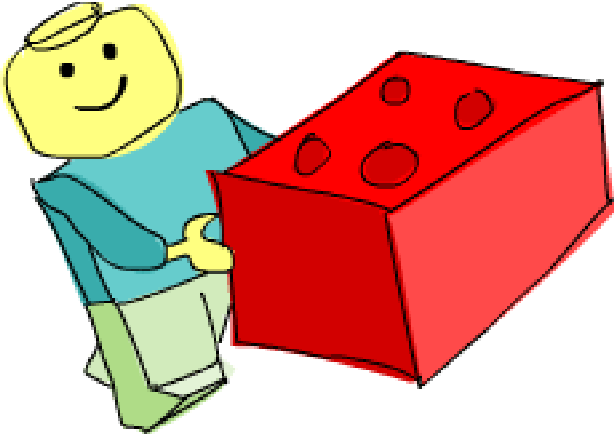
Christopher League

LIU Brooklyn

Northeast Scala Symposium

9 March 2012

What is a module?



What is a module?

WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia

Interaction

Help
About Wikipedia
Community portal
Recent changes
Contact Wikipedia

Toolbox

Print/export

Languages

Linguistic relativity

From Wikipedia, the free encyclopedia

The principle of **linguistic relativity** holds that the structure of a language affects the ways in which its speakers are able to conceptualize their world, i.e. their world view. Popularly known as the **Sapir–Whorf hypothesis**, or **Whorfianism**, the principle is often defined as having two versions: (i) the *strong* version that language determines thought and that linguistic categories limit and determine cognitive categories and (ii) the *weak* version that linguistic categories and usage influence thought and certain kinds of non-linguistic behavior.

The idea was first clearly expressed by 19th century thinkers, such as Wilhelm von Humboldt, who saw language as the expression of the spirit of a nation. The early 20th century school of American Anthropology headed by Franz Boas and Edward Sapir also embraced the idea. Sapir's student Benjamin Lee Whorf came to be seen as the primary proponent as a result of his published observations of how he perceived linguistic differences to have consequences in human cognition and behavior. Harry Hoijer, one of Sapir's students, introduced the term "Sapir–Whorf hypothesis",[1] even though the two scholars never actually advanced any such hypothesis.[2] Whorf's principle of linguistic relativity was reformulated as a testable hypothesis by

What is a module?

1. Separate compilation
2. Namespace management
3. Hiding / abstraction

Separate compilation in C

- "module" == file

```
#include <stdio.h>
```

- declarations

```
extern int foo;
```

- vs. definitions

```
int foo = 40;
```

Namespace management in C

- Hiding only, to limit namespace pollution

```
static void dont_export_me_bro()
{
    //...
}
```

Namespace management in C++

- Nesting, scope operator, imports, limited renaming

```
namespace greenfield { }
using namespace std;
using greenfield::cout;
namespace magick = future::tech;
magick::dwim();
```

Abstract data types in C

- Opaque type declaration ($\exists$), `void*` ($\forall$?)

```
struct stack;
stack* new_stack();
void push (stack*, void*);
```

Type abstraction in C++

- Generics (∀ T such that ???)

```
template<class T>
T& findMin(T* array, int size);
```

Hiding in C++

- Member access control

```
class Foo {
private:
  int x;
};
```

Hiding in C++

- Privacy via subsumption

```
struct stack {
    virtual void push(int) = 0;
    virtual int pop() = 0;
    static stack* create();
};
struct stackImpl : public stack {
    int a[SIZE];
    int k;
    // ...
};
```

Modules in Haskell

"Haskell's module design is relatively conservative"

— *A Gentle Introduction to Haskell*

```
module Utility.StatFS(
    FileSystemStats(..),
    getFileSystemStats) where

import Data.ByteString
import Data.ByteString.Char8 (pack)
import qualified Foreign as F

getFileSystemStats ::
    String -> IO (Maybe FileSystemStats)
getFileSystemStats path = {- ... -}

data FileSystemStats = {- ... -}
```

Type classes in Haskell

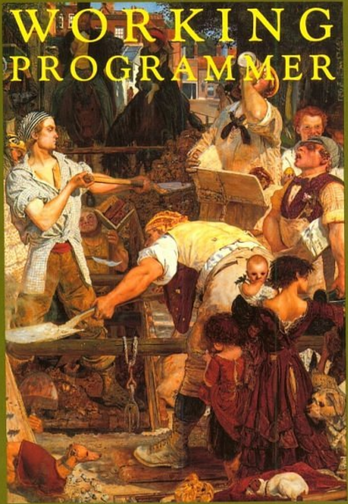
```
class Arbitrary a where
  arbitrary :: Gen a

instance Arbitrary Bool where
  arbitrary = choose (False,True)

instance (Arbitrary a, Arbitrary b) => Arbitrary (a,b)
  where
    arbitrary = liftM2 (,) arbitrary arbitrary

prn :: (Arbitrary a, Show a) => a -> IO ()
```

ML for the WORKING PROGRAMMER



2ND EDITION

L.C. Paulson

Signatures

- Type declarations and value specifications

```
signature COLLECTION = sig
    type 'a t
    val empty: 'a t
    val isEmpty: 'a t -> bool
end
signature QUEUE = sig
    type 'a t
    val enqueue: 'a * 'a t -> 'a t
    val dequeue: 'a t -> ('a t * 'a) option
end
signature DEQUE = sig
    include COLLECTION
    structure Front: QUEUE where type 'a t = 'a t
    structure Rear: QUEUE where type 'a t = 'a t
end
```

Structures

- Nested collections of defs, constrained by sigs

```
structure Deque :> DEQUE = struct
    type 'a t = 'a list * 'a list
    val empty = (nil, nil)
    fun isEmpty (nil, nil) = true
      | isEmpty _ = false
    structure Front = struct
        type 'a t = 'a t
        fun enqueue (x, (rs,fs)) = (rs, x::fs)
        fun dequeue (nil, nil) = NONE
          | dequeue (rs, x::fs) = SOME ((rs,fs), x)
          | dequeue (rs, nil) = dequeue (nil, rev rs)
    end
    structure Rear = struct (* ... *) end
end
```

Functors

- Structures parameterized by structures
- **Not** the thing from category theory, Haskell

```
functor TestDeque(D: DEQUE) = struct
    val q1 = D.empty
    val q2 = D.Front.enqueue (3, q1)
    val q3 = D.Front.enqueue (2, q2)
    val q4 = D.Rear.enqueue (4, q3)
    val q5 = D.Rear.enqueue (5, q4)
    (* ... *)
end

structure T = TestDeque(Deque)
```

‘Functorized’ style in ML

- Lift most structure dependencies to functor parameters

```
functor CompileF(M : CODEGENERATOR): COMPILE0 = ...
functor EvalLoopF(Compile: TOP_COMPILE) : EVALLOOP = ...
functor Interact(EvalLoop : EVALLOOP) : INTERACT = ...
```

- Instantiate dependencies at ‘link time’

```
structure Interact =
  Interact(EvalLoopF(CompileF(X86MC)))
```

A signature for mutable graphs

- Parameterize by type representing Vertex, Edge.

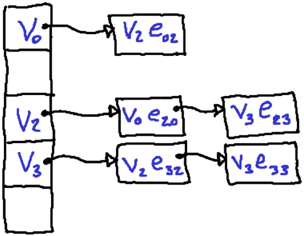
```
trait DirectedGraphSig {
  trait Graph[V,E] {
    def vertices: Iterator[V]
    def successors(v: V): Iterator[(V,E)]
    def add(v: V)
    def contains(v: V): Boolean
    def add(v1: V, v2: V, e: E)
    def get(v1: V, v2: V): Option[E]
  }
  def create[V,E]: Graph[V,E]
}
```

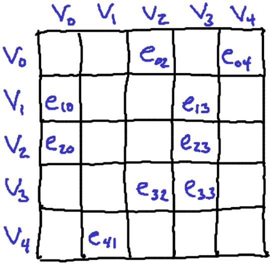
Yes, mutable — sorry

```scala
import scala.collection.generic.MutableMapFactory
import scala.collection.generic.MutableSetFactory
import scala.collection.mutable._
```

Representing graphs

- Adjacency list vs. adjacency matrix





- In general: $V \to V \to E$

Building graphs from maps

```
class DirectedGraphFun[
  M1[A,B] <: Map[A,B] with MapLike[A,B,M1[A,B]],
  M2[A,B] <: Map[A,B] with MapLike[A,B,M2[A,B]]]
 (MF1: MutableMapFactory[M1],
  MF2: MutableMapFactory[M2])
extends DirectedGraphSig
{
  class GraphImpl[V,E] extends Graph[V,E] {
    private val rep: M1[V,M2[V,E]] = MF1.empty
    // ...
  }
  def create[V,E] = new GraphImpl[V,E]
}
```

Instantiating the 'functor'

```
object AdjacencyList
extends DirectedGraphFun[
  HashMap, ListMap](HashMap, ListMap)

object AdjacencyMatrix
extends DirectedGraphFun[
  HashMap, HashMap](HashMap, HashMap)
```

- Easily build new modules with different space-time characteristics

Inspiration from C++ STL

```
#include <algorithm>
```

Graph search implementation

```
class GraphSearchFun[
  S[A] <: Set[A] with SetLike[A,S[A]]]
  (S: MutableSetFactory[S],
   WL: WorkListSig)
{
  type Path[V,E] = List[(V,E)]
  def search[V,E](g: DirectedGraphSig#Graph[V,E],
                  origin: V,
                  f: (V, Path[V,E]) => Unit) {
    val visited = S.empty[V]
    val work = WL.create[(V,Path[V,E])]
    work.put((origin, Nil))
    while(!work.isEmpty) {
      val (v1, path1) = work.take
      // ...
    }
}
```

Graph search relies on a work list

```
trait WorkListSig {
  trait WorkList[T] {
    def isEmpty: Boolean
    def put(x: T)
    def take: T
  }
  def create[T]: WorkList[T]
}
```

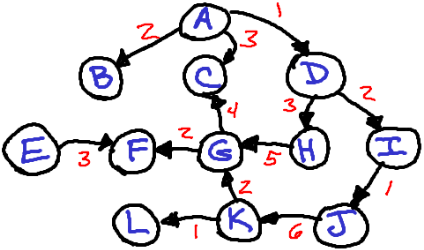
Various work list strategies

```
object LIFO extends WorkListSig {
  trait StackAsWorkList[T]
  extends Stack[T] with WorkList[T] {
    def put(x: T) = push(x)
    def take: T = pop
  }
  def create[T] = new Stack[T] with StackAsWorkList[T]
}
object FIFO extends WorkListSig {
  trait QueueAsWorkList[T]
  extends Queue[T] with WorkList[T] {
    def put(x: T) = enqueue(x)
    def take: T = dequeue
  }
  def create[T] = new Queue[T] with QueueAsWorkList[T]
}
```

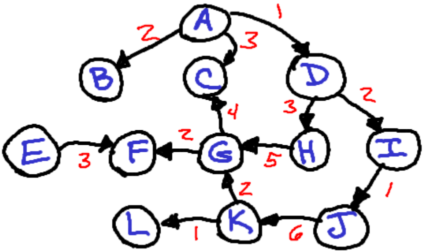
Voilà — different search algorithms

```
object BFS
extends GraphSearchFun[Set](Set, FIFO)

object DFS
extends GraphSearchFun[Set](Set, LIFO)
```




```
class ExampleFun[G <: DirectedGraphSig](G: G) {
  def example: G#Graph[String,Int] = {
    val g = G.create[String,Int]
    g.add(”A”, ”B”, 2); g.add(”A”, ”C”, 3)
    g.add(”A”, ”D”, 1); g.add(”G”, ”C”, 4)
    g.add(”D”, ”H”, 3); g.add(”D”, ”I”, 2)
    g.add(”E”, ”F”, 3); g.add(”G”, ”F”, 2)
    g.add(”H”, ”G”, 5); g.add(”I”, ”J”, 1)
    g.add(”J”, ”K”, 6); g.add(”K”, ”G”, 2)
    g.add(”K”, ”L”, 1)
    g
  }
}
```

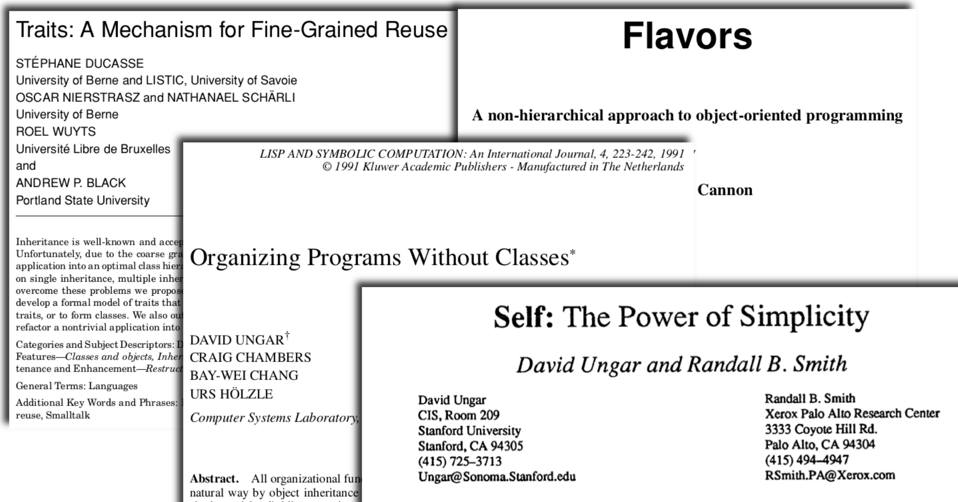



```
BFS on AdjacencyList
A from List()
D from List((A,1))
C from List((A,3))
B from List((A,2))
I from List((D,2), (A,1))
H from List((D,3), (A,1))
J from List((I,1), (D,2), (A,1))
G from List((H,5), (D,3), (A,1))
K from List((J,6), (I,1), (D,2), (A,1))
F from List((G,2), (H,5), (D,3), (A,1))
L from List((K,1), (J,6), (I,1), (D,2), (A,1))
```

```
DFS on AdjacencyMatrix
A from List()
B from List((A,2))
D from List((A,1))
I from List((D,2), (A,1))
J from List((I,1), (D,2), (A,1))
K from List((J,6), (I,1), (D,2), (A,1))
G from List((K,2), (J,6), (I,1), (D,2), (A,1))
F from List((G,2), (K,2), (J,6), (I,1), (D,2), (A,1))
C from List((G,4), (K,2), (J,6), (I,1), (D,2), (A,1))
L from List((K,1), (J,6), (I,1), (D,2), (A,1))
H from List((D,3), (A,1))
```

Untyped → typed

- Traits from Smalltalk, Self
- Flavors, mixins from CLOS



Room 101

A place to be (re)educated in Newspeak

Sunday, June 05, 2011

Types are Anti-Modular

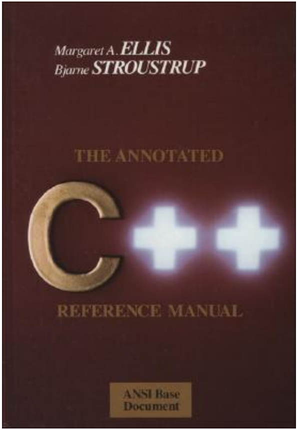
Last week I attended a workshop on language design. I made the off-the-cuff remark that types are actually anti-modular, and that comment resonated enough that I decided to tweet it. This prompted some questions, tweets being a less than perfect format for elaborate explanation of ideas (tweets are anti-communicative?). And so, I decided to expand on this in a blog post.

Saying that types are anti-modular doesn't mean that types are bad (though it certainly isn't a good thing). Types have pros and cons, and this is one of the cons. Anyway, I should explain what I mean and how I justify it.

The specific point I was discussing when I made this comment was the distinction between *separate compilation* and *independent compilation*. Separate compilation allows you to compile parts of a program separately from other parts. I would say it was a necessary, but not sufficient, requirement for modularity.

Untyped $\longrightarrow$ typed

"It was not obvious how to combine the C++ strong static type checking with a scheme flexible enough to support directly the 'mixin' style of programming used in some LISP dialects."



Untyped → typed ?

IEEE Software

The Larch Family of Specification Languages

John V. Guttag, Massachusetts Institute of Technology
James J. Horning, Digital Equipment Corporation
Jeannette M. Wing, Carnegie-Mellon University

The Larch Shared Language

The *trait* is the basic unit of specification in the Larch Shared Language. A trait introduces operators and specifies their properties. Sometimes the collection of operators will correspond to an abstract data type. Frequently, however, it is useful to define properties that do not fully characterize a type.

```
TableSpec: trait
  introduces
    new: → Table
    add: Table, Index, Val → Table
    # ∈ # : Index, Table → Bool
    eval: Table, Index → Val
    isEmpty: Table → Bool
    size: Table → Card
  constrains new, add, ∈, eval, isEmpty,
      size so that
    for all [ind, indl: Index,
      val: Val, t: Table]
      eval(add(t, ind, val), indl) =
        if ind = indl
        then val
        else eval(t, indl)
      ind ∈ new = false
      ind ∈ add(t, indl, val) =
        (ind = indl) | (ind ∈ t)
      size(new) = 0
      size(add(t, ind, val)) = if ind ∈ t
        then size(t) else size(t) + 1
      isEmpty(t) = (size(t) = 0)
```

Thanks!

`league@contrapunctus.net`
`@chrisleague`

- Code and slides will be made available later