Evolving Languages

(5 Steps to Go)

charles.forsyth@gmail.com

This talk didn't accompany a paper, which makes the slides harder to follow on their own. Although the talk might be available in video form, it might be useful to have something that can just be read. After the talk, I've added speaker's notes to help.

The talk was prompted by the 10th anniversary of Go's public release. Originally it was to be on Limbo, but I thought a talk on a (now) obscure predecessor, though easy to do, might not be all that valuable. Coincidentally I read a section in Kernighan's History of Unix that put Limbo into a sequence of languages on the way to Go. I had a slightly different view of it, not as a strict historic sequence but as the result of design choices prompted by the systems or environments in which the languages were to be used. After I'd started preparing the talk, I discovered a talk by Rob Pike that does look at the languages as more of an incremental development (see http://go-lang.cat-v.org/talks/slides/emerging-languages-camp-2010.pdf) and that can be read for contrast.

History

1985 CSP book “Communicating Sequential Processes”: Hoare (esp Occam)

1985 Squeak (“A Language for Communicating with Mice”): Cardelli & Pike

1992 Alef (Plan 9): Winterbottom

1994 Newsqueak (“A Language for Communicating with Mice”): Pike

1996 Limbo (Inferno): Dorward, Pike & Winterbottom (Lucent); Vita Nuova *

2009 Go: Griesemer, Pike & Thompson (Google) *

There were two versions of Hoare’s Communicating Sequential Processes. In the first version (1978?) communication was directly between named processes. A revision introduced communication through “channels” (unbuffered synchronising variables), a change that greatly improves modularity and effectiveness. A few languages adopted one or other of the CSP models. One set is relevant here. The 5 languages are Squeak, Alef, Newsqueak, Limbo and Go. They all feature concurrent processes communicating by sending and receiving values on channels. All but Squeak have roughly similar syntax (with many differences of detail). Notably, Rob Pike was involved directly in the design and implementation of 4 of them, and used all 5.

Squeak

```
proc Mouse = DN? . M?p . moveTo!p . UP? . Mouse

proc Kbd(s) = K?c .
        if c==NewLine then
                typed!s . Kbd(emptyString)
        else
                Kbd(append(s, c))
        fi

proc Text(p) = < moveTo?p . Text(p)
                :: typed?s . {drawString(s, p)}? . Text(p) >

type = Mouse & Kbd(emptyString) & Text(nullPt)
```

Squeak is not a complete language. It was intended to allow small UI components to be expressed using the CSP model instead of “callbacks” and “event loops”. The process structure and communication is part of Squeak, but fragments of C code appear within {} (much as in Yacc). The system is compiled to a state machine reflecting possible interleavings. In this example from the Squeak paper, there are three processes, connected by channels DN, UP, M, K, moveTo and typed. The ? and ! operators come from CSP: c?v receives a value from channel c; c!v sends a value on channel c. <> is a process list. P :: Q is non-deterministic choice between processes P and Q. The program “type” combines the processes to produce a program that puts complete lines of text on the screen at the place the mouse was last clicked.

Squeak was a small research experiment in writing user interface components using the CSP model, hence the focus on processes and communication, not a full language.

Alef

```
void
main(void)
{
	int r;
	Mevent m;
	chan(int) kbd;
	chan(Mevent) mouse;
	alloc kbd, mouse;
	proc kbdproc(kbd), mouseproc(mouse);
	for(;;) {
		alt {
		case r = <-kbd:
			if(r < 0)
				terminate(nil);
			/* process keyboard input */
			break;
		case m = <-mouse:
			/* process mouse event */
			break;
		}
	}
}
```

adt spec/body
adt[T]
tuple(int, byte*, int)
poly type (box/unbox, type case)

```
adt Point
{
	int	x; /* Access only by member fns */
	extern	int	y; /* by everybody */

		Point set(Point*); /* by everybody */
	intern	Point tst(Point);  /* only by member fns */
};

adt Stack[T] { ... T *data; ... };
```

rescue/raise [local error recovery blocks]
alloc/**unalloc**

Alef was a language for concurrent *systems* programming in the Plan 9 operating system. The type system and syntax are close to C. The panel to the right shows some of the new features: an **adt** associates functions with data, separating spec and body; a tuple type; two forms of polymorphism; and recovery blocks. The more important additions were explicit processes and tasks communicating through **chan** values. Processes are Plan 9 processes; tasks are coroutines (cooperative) within a process, but both procs and tasks communicate through channels. When a process suspends, all its tasks do too. Conversely, within a process, tasks guarantee mutual exclusion, and avoid the need for explicit locks, whereas processes are concurrent.

Alef was fully compiled. It was used to write a network stack, and the editing systems help and Acme (in Plan 9). I wrote my own applications in it, including a USENET (NNTP) server.

Disadvantages: NO gc, NO safety. Lack of gc (as Pike also notes) made concurrent programming more difficult, with manual resource control as in C. Alef was finally replaced by Plan 9 library libthread. Lesson: gc is desirable, as are immutable values.

Alef (cont'd)

```
void
mouseproc(chan(Mevent) mc, chan(int) termc)
{
   int fd, n;
   byte buf[1024];
   Mevent m;
   mousepid = getpid();
   fd = open("/dev/mouse", OREAD);
   for(;;) {
      n = read(fd, buf, sizeof(buf));
      if(n <= 0) {
         termc <-= 1;
         return;
      }
      m = decodemouse(buf, n);
      mc <-= m;
   }
}
```

Alef showed a way to link conventional OS primitives to processes+channels. The “mouseproc” above is a characteristic process type used by later languages: a simple loop converts a sequence of read system calls into a sequence of values sent on a channel. Similarly, a process can convert values received on a channel into write system calls. The channels are passed as parameters, making the process a general component for the start (end) of a pipeline, or to feed a multiplexor using **alt** to select non-deterministically between keyboard and mouse channels. Another idiom is passing in a channel termc to announce end of file.

In Alef, this must be a process not a task, since the read system call will block the process.

Newsqueak

```
rcM:=prog(n:NSIO, M:chan of Mouse, m:Mouse, inctl,outctl:Strchan){
        a:array of char;
        wdata:=mk(Strchan);
        for(;;)
                select{
                case m=<-M:
                        wdata=n.wdata;
                case wdata<-=MousetoB(m):
                        wdata=mk(Strchan);
                case a=<-n.rctl:
                        ;
                case <-n.rdata:
                        ;
                case a=<-inctl:
                        outctl<-=a;
                        switch(a){
                        case "shutdown":
                                n.wctl<-="remove";
                                become unit;
                        case "size":
                                a=<-inctl;
                                outctl<-=a;
                        }
                }
};
```

applicative language, immutable (c-o-w) values used to write complete window system

{chan of Mouse, chan of char, chan of array of char}

recursive structure: ws(M, K, Ctl):
  begin win(M', K', Ctl') to run window app, where
  ' versions are filtered by ws, restricting contents to those in that window
but ws itself matches signature

Newsqueak looked back to the application of CSP ideas in user interfaces, but this time using a full language, featuring not just garbage collection but immutable values. It was used to write a small but plausible window system, enough to manage overlapping layers on the screen, and run a shell or graphics in separate windows.

The novelty was to structure the system as a concurrent processes in a hierarchy, each taking (chan of Mouse, chan of char, chan of array of char) as input, multiplex and filter those channels and pass them on. Since the window system itself satisfies the interface, the window system can be run within a window, recursively, an idea that reappeared in later windowing systems for Plan 9 and Inferno.

Why applicative? Similar reasons to Erlang: concurrency is easier if processes don't share memory and values are immutable. Newsqueak also introduced type infererence for declarations, with : = as declaration & initialisation, used by Limbo & Go.

Diversion: Plan 9

resources represented by little directories (name space, file system)
computable name space:

- per-process granularity (local naming, not global)
- **bind**, **mount**, **unmount** operations (not privileged)
- build and compose a name space for an app (isolation)

9P file service protocol to implement all services
/net/tcp/clone → **/net/tcp/0/data /net/tcp/0/ctl /net/tcp/0/local /net/tcp/0/remote**
/net/dns → *write* **www.google.com**
→ *read* **www.google.com ip 216.58.210.196**
www.google.com ip 2a00:1450:4009:81a::2004
/net/cs → *write* **tcp!www.google.com!http**, *read* recipes to get there
→*read* **/net/tcp/clone 216.58.210.196!80**
/net/tcp/clone 2a00:1450:4009:81a::2004!80

In a short interlude, to understand certain aspects of Limbo, it's helpful to know a few things about the operating system Plan 9 from Bell Labs, specifically its representation of resources as name spaces, which can be built dynamically to per-process granularity. Even the networking interfaces are represented that way, as for TCP/IP, DNS and a general name service (connection service) here. Consequently, Plan 9 had relatively few system calls: open, read, write, close were the primary interfaces. (Aside: publish/subscribe? Open a name & read, blocking until the publishing server replies.)

Diversion: Inferno

Small operating system, runs native, or hosted (Linux, Win, Solaris, MacOS, 9, ...)

Simplicity and portability (x86, ARM, PowerPC, SPARC, MIPS, 680x0)

Same interface everywhere for its programs

Full OS: processes, shell, commands

Ideas of Plan 9 applied everywhere: resources and apps rep by name spaces; 9P

/prog; **/net/tcp**, **/net/dns**, **/net/cs**; **export**; **import /net**; **import /dev/draw**

Dis abstract machine (memory-memory, not stack-based); interpreter & JIT

Limbo concurrent programming language

After Plan 9, in 1996 Bell Labs (Lucent) developed a system called Inferno, originally for set-top boxes, later networked devices large & small. Inferno was acquired by Vita Nuova (York) in 2000. It addressed application portability not by defining a large set of conventional interfaces in library/package form to all existing resources, but by reproducing the Plan 9 model both as a native operating system *and* "hosted" by other operating systems (Windows, Solaris, Linux, MacOSX, etc.), allowing open/read/write/close to work its charm. For instance, networking (TCP/IP, DNS) presented the same interfaces on Inferno on all platforms as one saw on Plan 9.

Other aspects of portability were addressed by defining an abstract machine (*Dis*) with a portable executable format (Dis instructions), that could either be interpreted or compiled into target machine instructions by a simple JIT compiler. A new programming language called Limbo was used for both application and systems code (outside the kernel).

Limbo

type inference (as Newsqueak)

a: int = 3; → **a := 3;**

module-oriented:

M: module{...} spec

implement M; implementation

m := load M M->PATH;

***adt-list*: import m;**

string is *value*, array of *runes*

s := "Hello 😊"; a := array of byte s; s[6]==')😊'; t := string a; s != nil; s != ""; s[i:j]

int real big
(int, ref T, string) (tuples)
adt adt[T] adt[T] for { T => add: fn(...); }
adt methods ("**self**")
adt pick variants with **pick** stmt to inspect
ref T pointer to heap **ref fn** (mod, fn) pair
array of T reference type
list of T hd l tl l v :: l v :: nil
slices: a[i:j] a[i:] a[:j] len a a[i:j] = b[k:]
string s[i] s[i:j] len(s) s+t array of byte s
chan of T **chan[N] of T**
module types and values **load M** *pathname*

cheap processes (**spawn** server(reqs, ctl))
channels for communications:
<-c *(recv)* c <-= v *(send)* (i, v) = <-ca (array recv)

exceptions following CLU (**fn f() T raises E**)
exception handlers

compiles to abstract machine (Dis); JIT
dynamic linkage (load/unload modules)

Limbo has aspects of both Alef and Newsqueak. It is an imperative language, but has garbage collection, and some types with immutable values. I sometimes say the statement and expression syntax is C with the bugs fixed, and the type notation is Pascal with the bugs fixed. It is a safe language. (Aside: Dis itself is unsafe). The panel on the right lists some characteristic aspects of the language.

Its scheme for modularity is unusual: a module has a specification (signatures of the functions and data provided by the module) and one *or more* implementations. Implementations are loaded dynamically, selected by a path name in the name space of the current process (remember Plan 9 & Inferno). Modules are unloaded by nil'ing out the last reference. The language guarantees that resources are reclaimed when the last reference goes away. The garbage collector uses both reference counts and an incremental real-time collector as backup in case of cycles.

It also provides "cheap" processes: Limbo processes are automatically multiplexed (by the Inferno level) across heavyweight processes of the underlying OS.

Limbo runs only in the Inferno operating system, so its system interface is lean. Because Inferno runs hosted, Limbo applications can run under many operating systems, using that lean interface.

Limbo: module-oriented

```
Points: module {
   PATH: con "/dis/lib/points.dis";
   Point: adt {
      x, y: int;
      add: fn(p: self Point, q: Point): Point;
      text: fn(p: self Point): string;
   };
};

implement Points;
Point.add(a: self Point, b: Point): Point {
  return Point(a.x+b.x, a.y+b.y);
}
Point.text(p: self Point): string {
  return "("+string(p.x)+","+string(p.y)+")";
}
```

dynamically loaded/unloaded
can be >1 implementation (*PATH*)

```
implement Cmd;

include "sys.m";
include "points.m";

sys: Sys;
points: Points; Point: import points;

init(nil: list of string){
        sys := load Sys Sys->PATH;
        points := load Points Points->PATH;
        a := Point(1, 2);
        b := a.add(a);
        sys->print("%s\n", b.text()); # m->f
        …
        points = nil;              # unload
}
```

Here's a small example of a Limbo module to give the flavour.
The load operator can load any implementation that satisfies a given specification (the underlying implementation might do more).

There are 160 library modules including system interfaces (system calls, bitmap graphics, strings, lists, cryptography, networking, security, …)

Limbo: system interfaces

```
Sys: module {
   PATH: con "$Sys";
   …  41 system calls/library fns
   # File IO structures returned from file2chan
   # read:  (offset, bytes, fid, chan)
   # write: (offset, data, fid, chan)
   #
   Rread:    type chan of (array of byte, string);
   Rwrite:   type chan of (int, string);
   FileIO: adt {
      read:   chan of (int, int, int, Rread);
      write:  chan of (int, array of byte, int, Rwrite);
   };
   file2chan: fn(dir, file: string): ref FileIO;
};
sys := load Sys Sys->PATH (PATH=="$Sys")
sys->print("hello, %s\n", " ☺ ");
pid := sys->pctl(0, nil);   # get process ID
```

```
io := sys->file2chan("/chan", "service");
```

host A: **export /chan/service**

host B: **import hostA /chan/service**

host B: open /chan/service
read/write to access host A's svc

publish & subscribe → open/write/read

example: ramfile.b

Here is an interesting part of the system interface: the file2chan system call.
It puts a name in the process name space, and when another process opens it, its read and write requests are converted into messages (tuples) on a pair of channels.
Each tuple includes a private reply channel (an idiom in this style of CSP).
The reply on the channel is converted back into the result of the system call.
Since Inferno implements Plan 9's file service protocol 9P, that in turn allows a channel-oriented application to make its services available on a network for export and import. See the ramfile.b code for a simple Limbo application usingf file2chan.

Go

```
type Block interface {
        BlockSize() int
        Encrypt(src, dst []byte)
        Decrypt(src, dst []byte)
}

func encrypt(alg Block, src, dst []byte) int {
        alg.Encrypt(src, dst)
        return alg.BlockSize()
}

type MyString string  // define new type like string

func (s MyString) Odd() string {  // add Odd operation
        return "☹"
}

func g(){
        mys := MyString("hello, world")
        // mys[0] = 'x'   // illegal: strings immutable
        fmt.Printf("%v %v %v\n", mys.Odd(), mys, mys[2])
}
```

struct (composition)
array [N]T (value), slice a[i:j] (reference) a[i:j:max]
pointers (*T)
map[K] V v, ok := m[k]
string s[i] s[i:j] len(s) []rune(s) s+t
interface **interface{}**
type definitions (similar to Ada derived types); methods
error

goroutines (**go** f(...))
closures
defer
for i, v := **range** e {} # array/slice e, string
for k, v := **range** m {} # map m
for v := **range** c {} # chan c

func close(c chan)
func panic(v interface{})
func recover() interface{}

optimising compiler
explicit linkage, explicit cost

Once again a few snippets of code try to give the flavour of Go, with a list of characterstic features on the right. One novelty is Go's *interface* type, which accepts the value of any type that satisfies the interface. The language emphasises composition, not inheritance.

"Goroutines" provide an equivalent to "cheap processes" multiplexing goroutines across proxies for CPU cores, where the "goroutine" label emphasises the coroutine-like scheduling regime on each logical core.

Go emphasises speed of compilation and efficiency of execution. It is statically compiled. Linkage is static. It is the only one in this set that has a conventional optimising compiler. At the language level, it avoids primitives with a small notation that might have a large cost. For example, Limbo allows a receive from an array of channels as a form of non-deterministic receive, but Go does not. Strings are arrays of bytes, not arrays of Unicode code points and are indexed as arrays of bytes. Structures such as map are not safe for arbitrary concurrent access, to avoid imposing the cost of locks on all maps.

Compare ...

Similarities

C-like statements; CSP-derived concurrency (processes & channels);
many similar types (array, string; aggregate; chan of T; scalars)

Differences

Alef: compiled; C linking model; no gc; low-level control as in C; procs & tasks
poly/typeof; adt (spec/body); adt[T]; tuples; raise/recover

Limbo: abstract machine (but JIT); hybrid gc (ref/mark); procs; error-handling; adt
modules; separate spec/implementation; dynamic loading/unload; ref; adt[T]
no existentials; list of T; array as ref type; string of runes; **cyclic**

Go: optimising compiler; gc; goroutines; interfaces; interface{}; pointers; map[K] V
packages (static linking, naming); array as value; string of bytes

Evolution of error-handling

There are many similarities, even where there are differences of detail.
The unifying theme is concurrency using the CSP model.

Other differences are more than detail: static or dynamic compilation; static or dynamic linkage; multiple implementations of a module vs single instance of a package; are garbage collection rules guaranteed by the language or not?

Bigger differences between languages are those features that couldn't be removed, or couldn't be transplanted into one of the others without significantly changing it. For example, the map type could easily be added to Limbo, but dynamic modules and the spec/body distinction allowing several implementations selected at run-time is an essential feature of the language. Similarly, interfaces are essential to Go.

Even so, there is some evolution across the set. Let's look at error handling.

Differences

~~concurrency~~
~~statements~~, ~~expressions~~, ~~types~~
allocation
compilation
composition
polymorphism
existentials
modularity
portability
error-handling
safety

Not used in the talk. Left as notes.

CSP inspired, similar syntax, but ...

all but Squeak allow chan as first-class type and values, allowing chan over chan (cf. Milner's π-calculus)

type systems (struct/adt, poly, adt[T], tuples, modules, receivers, interface)

same types, different detail (eg, strings, array/slice, spec/body, dynamic modules, composition)

other detail (<-array, slice assignment, built-in/external linkage, process, task, goroutine, rescue/defer)

interpreted or compiled? static or dynamic? portability?

storage management (gc or explicit? ref-counted? cyclic. immutable list of T, string, slice isolation)

system interface (special, library, modules, virtual OS, packages), existing libraries, **error-handling**

environment (lab, terminal, systems programming, embedded/server, large-scale)

Not used in the talk. Left as notes, considering various choices made in various aspects of the languages.

not just gc but immutable values (cf. Erlang), but array of list of T in Limbo. atomic values. race conditions and memory model. partial values. concurrent collection. stacks. local functions.
par {...} in Alef. aggr. poly/zerox. alt/select and modularity. Alef named channels and distribution.

error handling

Alef:

```
alloc a, b;
rescue {
        unalloc a, b;
        return 0;
}
alloc c;
rescue {
        unalloc c;
        raise;
}
dostuff();
if(error)
        raise;
```

Limbo:

```
String: module{
    …
    toint: fn(s: string, base: int):
(int, string);
};

(val, err) := strings->toint(s, 10);
if(err != “”)
        fatal(“bad int: “+err);
```

Go:

```
type error interface {   // built-in
        Error() string
}
func read(name string) ([]byte, error) {
        file, header, err := r.FormFile(name)
        if err != nil {
            return nil, err
        }
        defer file.Close()
        return io.ReadAll(file), nil
}
```

```
package strconv
func ParseInt(s string, base int, bitSize int) (int64,
error)
```

```
s, err := strconv.ParseInt("-354634382", 10, 32)
if err != nil {
    return errors.Wrap(err, “illegal integer”)
}
```

Compared to other language features, we might see some direct evolution in error handling. Alef had rescue and raise to handle errors locally (ie, within a single function). They were not exceptions that propagated. Limbo introduced returning a *tuple* from a function: one or more components for the normal value, and a string that carried a diagnostic in case of error. Go extended that convention to add a distinguished (built in) **error** type, which is a Go interface, allowing programmers to implement custom handling of errors. Errors are handled using “normal” statements of the language. (It is an interesting aspect of Go.) Go also added the **defer** statement to associate operations (eg, clean-up) with the return from a function, including on error.

exception handling

Limbo (after CLU):

```
Syntax: exception(string);

readjson(fd: ref Iobuf): (ref JValue, string)
{
        {
                p := Parse.mk(fd);
                c := p.getns();
                if(c == Bufio->EOF)
                        return (nil, nil);
                p.unget(c);
                return (readval(p), nil);
        }exception e{
        Syntax =>
                return (nil, sys->sprint("JSON syntax error (offset %bd):
%s", fd.offset(), e));
        }
}
```

```
readval(p: ref Parse): ref JValue raises(Syntax)
{
        while((c := p.getc()) == ' ' || c == '\t' || c == '\n' || c == '\r')
                {}
        if(c < 0){
                if(c == Bufio->EOF)
                        raise Syntax("unexpected end-of-input");
                raise Syntax(sys->sprint("read error: %r"));
        }
        case c {
        '{' =>
                # parse object …
                return ref JValue.Object(obj);
        '[' =>
                # parse array recursively
                return ref JValue.Array(arr);
        * ⇒
                raise Syntax("unexpected character");
        }
        # not reached
}
```

After Vita Nuova acquired Limbo with Inferno, we took a more conventional approach to error handling. Originally Limbo had some system calls that had a similar effect to exceptions (think of Go's panic/recover), but were not known to the compiler and were unconnected to the syntax (eg, the scope of sys->raise and sys->recover could be obscure). It was error-prone and uncomfortable, and we decided to add exceptions, but kept close to the design of exceptions ("signals") for the CLU language. Unlike Ada, since the signature of a function includes its exceptions, there is a visible sign in the code and documentation. Following CLU, an unhandled exception is converted to *failure*, the same signal as for failure to allocate memory (eg for string operations), and nil pointer and bounds checks. Those checks can appear within expressions and can't always be made explicit (eg, out of stack space).

Now, I'd be inclined to replace the Limbo error tuple by a sum type with a Go-like error value as an alternative (eg, f(): int | error).

exception handling

Go:

```
func f(){
        defer func(){
                if p := recover(); p != nil {
                        // handle panic error val p
                        panic(p)      //propagate
                }
        }()
        g()
}

func g(){
        h()
}
func h(){
        panic("boom!")  // interface{} in general
}
```

As well as the special error interface and explicit checks, Go programs might also have to deal with run-time errors (eg, failure to allocate memory etc) that don't lend themselves to explicit checks. Its panic/recover built-ins provide a way to signal failure and recover from it, but outside the syntax of the language. It unwinds the stack to recovery point in essentially the same way as conventional exceptions, but uses the existing defer statement to avoid introducing special syntax.

Environments and design choices

environment

Squeak	lab
Alef	systems programming (network stack)
Newsqueak	terminal (“gnot”), proof of concept
Limbo	set-top box, embedded systems, supporting servers
Go	*large-scale* systems, emphasis on efficiency

system interface

Squeak	wrapper round C fragments, generated state machine
Alef	libraries (themselves in Alef); no access to C
Newsqueak	interpreter with interfaces to supporting C code
Limbo	Inferno virtual OS with modules to access existing libraries
Go	via packages including **cgo** to access existing libraries

choices

static? dynamic?
interpreted? compiled?
port. by abstract machine?
port. by cross-compilation?
language only? OS level?
strings
arrays
processes (Alef vs rest)
gc characteristics
(Alef :: Limbo :: Go)
error-handling

Are the differences arbitrary? Design choices (some shown in the panel to the right) are influenced by the environment in which the language is to be used, which also affects the system interface the language (run-time) provides.

While not precluding other target areas, Go needed to be competitive in Google’s large-scale systems, hence an emphasis on efficiency and scaling. Limbo emphasised control of module loading and unloading to reduce memory requirements on small machines and allow dynamic reconfiguration. It relies on the Inferno virtual OS infrastructure for its module loading primitive, and its portability and ease of distributed application development. Indeed, in process management and even in error handling, Inferno provides mechanisms outside the Limbo language, but accessible from it, that none of the other languages can provide.

The invariant has been the adherence to the CSP model for more than three decades, finally achieving widespread use in Go.

But serverless ...

Finally got CSP model into vastly wider use …

… only for the virtual machine to change!

How useful is CSP …

in a client-server HTTP system of spontaneous requests?

for “serverless” systems, with spontaneous *function* calls, etc?

How to compose intermittent and sporadic program snippets?

Extra-language composition languages proposed

for distributed state machines

Begin again …

… linguistic support for it? Abstract CSP (composition across time)? Also, Squeak for IoT?

Not for the first time, we’ve come to grips with the previous problem that hardware/software systems architects gave us, only to have them mutate it again. Reminiscent of Dijkstra’s paper about IBM 620, interrupts, etc
Now serverless has “at least once” not “at most once”, idempotent demands, cold/warm/hot state, databases, etc. What language and system support can salvage this?
At the other end of the scale, there are tiny machines that might benefit from the CSP ideas. Squeak for IoT?