Some Trucs and Machins about Google Go

Narbel

The Cremi's Saturdays, Saison I

University of Bordeaux 1

April 2010

(v.1.01)

A “New” Language in a Computerizationed World

- A magnificent quartet...:
  - Microsoft: Windows, Internet Explorer (Gazelle?), Microsoft Office, Windows Mobile, C#/F#.
  - Sun: Solaris, HotJava, StarOffice, SavaJe, Java.
  - Apple: MacOS, Safari, iWork, iPhoneOS, Objective-C.
  - Google: ChromeOS, Chrome, Google Docs, Android, Go.
- Go: the last brick (born in November 2009).

Position Apr 2010	Position Apr 2009	Delta in Position	Programming Language	Ratings Apr 2010	Delta Apr 2009	Status
1	2	↑	C	18.058%	+2.59%	A
2	1	↓	Java	18.051%	-1.29%	A
3	3	=	C++	9.707%	-1.03%	A
4	4	=	PHP	9.662%	-0.23%	A
5	5	=	(Visual) Basic	6.392%	-2.70%	A
6	7	↑	C#	4.435%	+0.38%	A
7	6	↓	Python	4.205%	-1.88%	A
8	9	↑	Perl	3.553%	+0.09%	A
9	11	↑↑	Delphi	2.715%	+0.44%	A
10	8	↓↓	JavaScript	2.469%	-1.21%	A
11	42	↑↑↑↑↑↑↑↑↑	Objective-C	2.288%	+2.15%	A
12	10	↓↓	Ruby	2.221%	-0.35%	A
13	14	↑	SAS	0.717%	-0.07%	A
14	12	↓↓	PL/SQL	0.710%	-0.38%	A
15	-	↑↑↑↑↑↑↑↑↑	Go	0.710%	+0.71%	A
16	15	↓	Pascal	0.648%	-0.07%	B
17	17	=	ABAP	0.625%	-0.03%	B
18	20	↑↑	MATLAB	0.616%	+0.13%	B
19	22	↑↑↑	ActionScript	0.545%	+0.09%	B
20	19	↓	Lua	0.521%	+0.03%	B

Objective-C and Go: a new kind of progression...(from `www.tiobe.com`, April 2010)

Creating New Languages and Tactics...

- "Securing" languages for oneself, a modern tactics? e.g.:
  - Java $\leftarrow$ C# (Microsoft).
  - OCaml $\leftarrow$ F# (Microsoft).
  - C $\leftarrow$ Go ? (Google).
- In the Go team, some "C/Unix-stars":
  - Ken Thompson (Multics, Unix, B, Plan 9, ed, UTF-8, etc. – Turing Award).
  - Rob Pike (Plan 9, Inferno, Limbo, UTF-8, etc.)

One of the Underlying Purposes of Go...!?

- Many recent successful languages are dynamic-oriented, i.e. Python, Ruby, etc. or extensions/avatars of Java, like Clojure and Groovy.
- In the official Go tutorial: "It feels like a dynamic language but has the speed and safety of a static language."
- Even if there exist dynamic languages with very efficient compilers (cf. CLOS), Go takes a step out of the current dynamic-oriented trend, and proposes more type-safe programming, while at once focussing on fast compilation and code execution... (i.e. towards high-performance web programming?).

Aparté: Compiled, Interpreted, Tomato-Souped...

- In "*A Tutorial for the Go Programming Language*", one finds **"Go is a compiled language"**... (no more meaning than "interpreted language"...).

- For instance, consider the Wikipedia entry about Python:
  - At first: "Python is an **interpreted**, interactive programming language created by Guido van Rossum."
  - Next (at 16:37, 16 Sept. 2006), the truth took over...: "Python is a **programming language** created by Guido van Rossum in 1990. Python is fully dynamically typed and uses automatic memory management." [...] "The de facto standard for the language is the CPython implementation, which is a **bytecode compiler and interpreter** [...] run the program using the Psyco **just-in-time compiler**."

References about Go

Bibliography about Go is still rather short...:

- Official doc (on `golang.org`):
  - *The Go Programming Language Specification*.
  - *Package Documentation* (the standard lib doc).
  - *A Tutorial for the Go Programming Language*.
  - *Effective Go* : the main text on the official site *(could be better...)*
  - Various FAQs and `groups.google.com/group/golang-nuts/`.
  - *The Go Programming Language*, slides of Rob Pike, 2009.
- A lot of bloggy internet stuff...

Yet Another Hello Hello

```
package main
import "fmt" // formatted I/O.

func main() {
    fmt.Printf("Hello Hello\n")
}
```

Echo Function in Go

```
package main
import (
    "os"
    "flag" // command line option parser
)

var omitNewline = flag.Bool("n", false, "no final newline")
const (
    Space = " "
    Newline = "\n" )

func main() {
    flag.Parse() // Scans the arg list and sets up flags
    var s string = ""
    for i := 0; i < flag.NArg(); i++ {
        if i > 0 {
            s += Space
        }
        s += flag.Arg(i)
    }
    if !*omitNewline {
        s += Newline
    }
    os.Stdout.WriteString(s)
}
```

Echo Function in C (not much of a difference...)

```
int
main(int argc, char *argv[]) {
    int nflag;

    if (*++argv && !strcmp(*argv, "-n")) {
        ++argv;
        nflag = 1;
    }
    else
        nflag = 0;

    while (*argv) {
        (void)printf("%s", *argv);
        if (*++argv)
            (void)putchar(' ');
    }
    if (nflag == 0)
        (void)putchar('\n');
    fflush(stdout);
    if (ferror(stdout))
        exit(1);
    exit(0);
}
```

Some Interesting Points about Go...

- A language should not in general be compared from its syntax, sugar gadgets, construct peculiarities...
  (we skip them).
- And even if Go does not include revolutionary ideas, there are some points worth to be discussed, e.g.:
  - Polymorphic functions (CLOS-Haskell-like).
  - Typed functional programming.
  - Easy to use multi-threading.

Functions

- Basic function header:

```
func <function name>(<typed args>) <return type> {}
```

- Basic function header with multiple returns:

```
func <function name>(<typed args>) (<return types>) {}
```

- For instance:

```
func f(int i) (int, string) {
    return (i+1) ("supergenial")
}

x, y := f(3);
```

Functions with Multiple Return Values

- A common use of functions with multiple return values: **error management**... (there are no exceptions in Go).
- For instance:

```
func ReadFull(r Reader, buf []byte) (n int, err os.Error) {
    for len(buf) > 0 && err == nil {
        var nr int;
        nr, err = r.Read(buf);
        n += nr;
        buf = buf[nr:len(buf)];
    }
    return;
}
```

Functions Associated to a Type

- Header for a function to be associated to a type T:

```
func (<arg> T) <functionName>(<typed args>) <return type> {}
```

The argument arg of type T can be used in the body of the function like the other arguments. For instance:

```
type Point struct {
  x, y float64
}

func (p Point) Norm() float64 {
  return math.Sqrt(p.x * p.x + p.y * p.y);
}

p := Point{1, 2};
[...] p.Norm() [...];
```

- $\Rightarrow$ Similar syntax to C++/Java: the associated type argument is privileged, as is an object rel. to its methods.

Interfaces : Towards Polymorphism

- An example of **interface**:

```
type Truc interface {
     F1() int;
     F2(int, int);
}
```

  $\Rightarrow$ Truc is then a registered type (in fact, a type of types).

- Rule: Every data type `T` with associated functions as declared in the interface `Truc` is **compatible with** `Truc`.
  $\Leftrightarrow$ Every instance of type `T` is also an instance of type `Truc`.

- $\Rightarrow$ A type can satisfy more than one interface.

Interfaces : Towards Polymorphism

```
type PrintableTruc interface {
    PrintStandard();
}

type T1 struct { a int; b float; c string; }
type T2 struct { x int; y int; }

func (t1 *T1) PrintStandard() {
    fmt.Printf("%d %g %q \n", t1.a, t1.b, t1.c);
}
func (t2 *T2) PrintStandard() {
    fmt.Printf("%d %d \n", t2.x, t2.y);
}

func F(p *PrintableTruc) {  // polymorphic
    p.PrintStandard();
}

func main() {
    nuple1 := &T1{ 7, -2.35, "blueberries" };
    nuple2 := &T2{ 1, 2};
    nuple1.PrintStandard();
    F(nuple1); // output: 7 -2.35 "blueberries"
    F(nuple2); // output: 1 2
}
```

Type compatibility and Upcasts

Type compatibility and type-safe implicit upcasting:

```
[...]
type T3 struct {d float}
[...]

var nuple PrintableTruc;
nuple1 := &T1{ 7, -2.35, "blueberries" };
nuple3 := &T3{ 1.43242 };

nuple = nuple1 // ok
nuple = nuple3; // statically not ok: *T3 is not PrintableTruc
```

Dynamic Selection

Function selection can be dynamic (and still type-safe):

```
func main() {
    var nuple PrintableTruc;
    nuple1 := &T1{ 7, -2.35, "blueberries" };
    nuple2 := &T2{ 1, 2};

    for i := 0 ; i < 10; i++ {
        if (rand.Int() %2) == 1 { nuple = nuple1 }
        else { nuple = nuple2 }
        nuple.PrintStandard();
    }
}
```

How to Use Existing Interfaces: a Basic Example

- In the package `sort`, there is an interface definition:

```
type Interface interface {
    Len() int
    Less(i, j int) bool
    Swap(i, j int)
}
```

(idiomatic: here, the complete name is `sort.Interface`).

- There is also a polymorphic function:

```
func Sort(data Interface) {
    for i := 1; i < data.Len(); i++ {
        for j := i; j > 0 && data.Less(j, j-1); j-- {
            data.Swap(j, j-1)
        }
    }
}
```

How to Use Existing Interfaces: a Basic Example

⇒ Using sorting for a user-defined type:

```
import ("fmt"
        "sort")

type IntSequence []int

//IntSequence as a sort.Interface:
func (p IntSequence) Len() int { return len(p) }
func (p IntSequence) Less(i, j int) bool { return p[i] < p[j] }
func (p IntSequence) Swap(i, j int) { p[i], p[j] = p[j], p[i] }

func main() {
    var data IntSequence = []int{1, 3, 4, -1, 5, 333}
    sort.Sort(data)      // data called as a sort.Interface
    fmt.Println(data);
}
```

Output:

```
[-1 1 3 4 5 333]
```

Maximum Polymorphism and Reflection

- In C: maximum polymorphism through `void*`.
  In Go: maximum polymorphism through the **empty interface**, i.e. "`interface {}`".
- For example, the printing functions in `fmt` use it.
- $\Rightarrow$ Need for some **reflection** mechanisms, i.e. ways to check at runtime that instances satisfy types, or are associated to functions.
- For instance, to check that `x0` satisfies the interface `I`:

```
x1, ok := x0.(I);
```

(`ok` is a boolean, and if true, `x1` is `x0` with type `I`)

Duck Typing

- Go functional polymorphism is a type-safe realization of "**duck typing**".
- Implicit Rule: If something can do this, then it can be used here.

  Naively: "When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck."
- $\Rightarrow$ Opportunistic behavior of the type instances.
- $\Rightarrow$ Dynamic OO languages like CLOS or Groovy include duck typing in a natural way.
- In static languages: duck typing is realized as a **structural typing mechanism** (instead of **nominal** in which all type compatibilities should be made explicit – see e.g., `implements`, `extends` in Java).

Go Interfaces and Structuration Levels

- Go interfaces ⇒ A type-safe overloading mechanism where *sets of overloaded functions* make type instances compatible or not to the available types (interfaces).

- The effect of an expression like:

  `x.F(..)`

  depends on all the available definitions of `F`, on the type of `x`, and on the set of available interfaces where `F` occurs.

- About the grain of structuration dilemma between the functional and modular levels: Go votes for the functional level, but less than CLOS, a little more than Haskell, and definitely more than Java/C# (where almost every type is implemented as an encapsulating class)...

Other Languages

- Go interface-based mechanism is not new, neither very powerful...
- For instance, Haskell **type classes**:

```
class SmallClass a where   // similar to a Go interface
  f1 :: a -> a
  f2 :: a -> Int

instance SmallClass Char where   // implementation
  f1 x =  chr ((ord x) + 1)
  f2 = ord

instance SmallClass Bool where   // another implementation
  f1 _ = True
  f2 True = 1
  f2 False = 0
```

- Haskell offers type inference with constrained genericity, and inheritance...

Other Languages

- Go structural-oriented type system is not new, neither very powerful...
- For instance, inferred Ocaml **open object types**:

```
# let f x =
    (x#f1 1) || (x#f2 'a');;

//val f : < f1 : int -> bool;
//          f2 : char -> bool; .. > -> bool
```

- OCaml offers type and interface inference with constrained genericity, and inheritance...

Interface Unions

- In Go, no explicit inheritance mechanism.
- The only possibility: some implicit behavior inheritance through **interface unions** (called "*embedding*"):

```
type Truc1 interface {
     F1() int;
}

type Truc2 interface {
     F2() int;
}

type Truc interface {
     Truc1;     // inclusionf
     Truc2;     // inclusion
}
```

$\Rightarrow$ Rule: If type `T` is compatible with `Truc`, it is compatible with `Truc1` and `Truc2` too.

Functional Programming

- Functional programming (FP) has two related characteristics:
  - **First-class functions**. Functions/methods are first-class citizens, i.e. they can be (0) named by a variable; (1) passed to a function as an argument; (2) returned from a function as a result; (3) stored in any kind of data structure.
  - **Closures**. Functions/methods definitions are associated to some/all of the environment when they are defined.
- When these are available $\Rightarrow$ Most of the FP techniques are possible
- Caveat: Only full closures can ensure *pure FP* (and in particular, *referential transparency*).

Go Functions are First-Class

```
type IntFun func(int) int;

func funpassing(f0 IntFun) {
   fmt.Printf("Passed : %d \n", f0(100));
}
func funproducer(i int) IntFun {
   return func(j int) int {return i+j;}
}

func main() {
  f := func (i int) int {return i + 1000;} //anonymous function
  funpassing(f);
  fmt.Printf(" %d %d \n", f(5), funproducer(100)(5));

  var funMap = map[string] IntFun {"f1": funproducer(1),
                                   "f2": funproducer(2)};
  fmt.Printf(" %d \n", funMap["f1"](5));
}
```

Output:

```
Passed : 1100
1005 105
6
```

Closures are Weak in Go

Go closures are not as strong as required by pure FP:

```
func main() {
  counter := 0;

  f1 := func (x int) int { counter += x; return counter };
  f2 := func (y int) int { counter += y; return counter };

  fmt.Printf(" %d \n", f1(1));
  fmt.Printf(" %d \n", f2(1));
  fmt.Printf(" %d \n", f1(1));
}
```

Output:

```
1
2
3 // => no referential transparency !
```

Concurrence

- The idea: to impose a sharing model where processes do not share anything implictly (cf. Hoare's CSP).
- Motto: "Do not communicate by sharing memory; instead, share memory by communicating."
- A consequence: to reduce the synchronization problems (sometimes at the expense of performance).
- Only two basic constructs:
  - "**Goroutines**" are similar to threads, coroutines, processes, (Google men claimed they are sufficiently different to give them a new name)
  - **Channels**: a typed FIFO-based mechanism to make goroutines communicate and synchronize.

Goroutines

- To spawn a goroutine from a function or an expression, just prefix it by `go` (in Limbo, it was `spawn`):

```
go f()
go func () { ...} // call with an anonymous function
go list.Sort()
```

  Goroutines are then automatically mapped to the OS host concurrency primitives (e.g. POSIX threads).

- NB: A goroutine does not return anything (side-effects are needed)

Channels

- Basics operation on channels:

```
ch := make(chan int) // initialization

/* Sending a value to a channel, blocked until it is read */
ch <- 333

/* Reading a value, blocked until a value is available */
i := <- ch
```

- $\Rightarrow$ Implicit synchronization through the channel semantics.

Concurrent Programming

Example of a simple goroutine without any synchronization (NB: recall that a goroutine cannot return a value):

```
var res int = 0;

func comput(from int, to int) int {
    tmp := 0
    for i := from; i < to; i++ {
        tmp += i
    }
  res = tmp
}

func main() {
    go comput(0, 1000000);
    time.Sleep(1000000);
    fmt.Println("The result:", res)
}
```

(without `time.Sleep`, the printed result will be 0...).

Concurrent Programming

Example of a simple goroutine with synchronization (using anonymous functions is the idiomatic way of calling expressions as a goroutine):

```
func comput(from int, to int) int {
    tmp := 0
    for i := from; i < to; i++ {
        tmp += i
    }
   return tmp
}

func main() {
    ch := make(chan int)
    go func(){ ch <- comput(0, 1000000) }()
    res := <- ch // blocked until the calculation is done
    fmt.Println("The result:", res)
}
```

Concurrent Programming

A producer/consumer agreement: (inspired from www.cowlark.com)

```
var ch chan int = make(chan int);

func prod() {
      counter := 0;
      for {
          fmt.Println("produced: ", counter)
          ch <- counter;
          counter++
       }
}
func consum() {
       for {
          res := <- ch;
          fmt.Println("consumed: ", res);
       }
}

func main() {
    wait := make(chan int)
    go prod();
    go consum();
    <- wait // wait to infinity...
}
```

Channel Behavior Customization

Customizing the blocking behavior is possible. In particular:

- By specifying a FIFO buffer size, that is, by using a **buffered channel**:
  - Sending data to such a channel blocks if the buffer is full.
  - Reading from such a channel will not block unless the buffer is empty.
- Declaration example:

```
ch := make(chan int, 100)
```

Parallelization with Channels

Computations with some parallelization: (inspired from *Effective Go*)

```
const NCPU=4
var ch chan int = make(chan int, NCPU)

func comput(from int, to int, ch chan int) {
    tmp := 0
    for i := from; i < to; i++ {
        tmp += i
    }
    ch <- tmp
}

func main() {
    iterat := 10000000;
    for i := 0; i < NCPU; i++ {
        go comput(i*iterat/NCPU, (i+1)*iterat/NCPU, ch)
    }

    tmp := 0;
    for i :=0; i < NCPU; i++ {
        tmp += <- ch;
    }
    fmt.Println("The result:", tmp)
}
```

Goroutines and FP

- Very naturally comes the idea of mixing multi-threading and functional programming (use of anonymous functions).

- At least two difficulties in doing that in Go:
  - Goroutines do not return any result ($\Rightarrow$ side-effects are needed).
  - Closures are weak.

- For instance, the following variation of the preceding example does not work:

```
for i := 0; i < NCPU; i++ {
    go func() {
        ch <- comput(i*iterat/NCPU, (i+1)*iterat/NCPU)
    }()
}
```

(`i` is shared by all the processes...)

A Partial Presentation...

Go has other points of interest which could have been presented and/or more discussed here:

- Reflection.
- The standard library.
- Memory management and pointers.
- Executable building.
- Error system (e.g., functions `panic`/`recover`, package unsafe, etc.),

Next Time...

The subject of the talk will be...:

- "D is a systems programming language. Its focus is on combining the power and high performance of C and C++ with the programmer productivity of modern languages like Ruby and Python." [..!?]
- "The D language is statically typed and compiles directly to machine code. It's multiparadigm, supporting many programming styles: imperative, object oriented, and metaprogramming. It's a member of the C syntax family, and its appearance is very similar to that of C++." [..!?]
- "It adds to the functionality of C++ by also implementing design by contract, unit testing, true modules, garbage collection, first class arrays, associative arrays, dynamic arrays, array slicing, nested functions, inner classes, closures, anonymous functions, compile time function execution, lazy evaluation and has a reengineered [..!?]"

Go Installation

- Go is quite easy to install (be careful about the four environment variables GOROOT, GOARCH, GOOS, GOBIN).
- Fully available for Linux, FreeBSD, Darwin.
  (NaCl : on its way)
  (Windows : on its way + partial Cygwin distribution).
- Needed : Mercurial (the VCS chosen by Go).
- To make a basic Go program be useful:
  - Source code file extension is ".go".
  - Compilation with "8g" $\Rightarrow$ object file with extension ".8".
  - Linking with "8l" $\Rightarrow$ exec file "8.out".
  - Replace the above "8"'s by "6"'s for 64 bits architectures.