The Good language manual

Summary

Good is a strongly typed general-purpose programming language that is designed

- to allow high code reusability with
  - generics
  - traits
- to allow using thousands of libraries and embed Good programs in C projects with
  - full C compatibility
- to allow high level programming with
  - algebraic data types
  - type inference
  - ownership
  - custom coercion
  - error system
- to allow low level programming with
  - manual memory management with raw pointers
  - full C compatibility

Basically the Good language allows everything one may need. The Good language is a good language.

Contents

1 Program structure

The Good program is a file that contains several `items`. An `item` can be

- a function
- an extern declaration
- a struct
- an enum
- a type alias
- a trait
- a trait implementation
- a constant

2 Functions

All (or almost all) the Good code is contained in functions:

```
fn one_two_three() {
  println("ready..");
  println("set...");
  println("go!");
}
```

Functions can be called from other functions:

```
fn print_something() {
  one_two_three();
}
```

Functions with just one expression (like the previous one) can be written in a short form:

```
fn print_something()
  do one_two_three()
```

The `main` function is special in that it is called at the beginning of the program and is required to be present in the code:

```
fn main()
  do println("program started (and finished)")
```

Functions can accept parameters:

```
fn print_sum(a: i32, b: i32)
  do println(a + b)
```

Here `i32` is a type of 32-bit integers.

To call such functions, one shall pass arguments of corresponding types to it:

```
fn main()
  do print_sum(123, 321)
```

Functions can also return some result:

```
fn add(a: i32, b: i32) i32
  do a + b
```

Here `i32` is a return type, and the last expression of a block or, in this case, the expression after `do` is returned.

The result of such functions can be used by the caller:

```
fn main()
  do println(add(123, 321))
```

Functions with at least one parameter can also be called with “dot-notation”:

```
fn main()
  do add(123, 321).println()
```

Here the `println` function is called with “dot-notation”, and `add(123, 321)` becomes its first argument. This expression is equivalent to the previous one.

The name of the parameter of a function can be inferred from its type:

```
fn double(i32) i32
  do i32 * 2
```

Here the name of a parameter is `i32` and the type is also `i32`. More on type-to-name inference in Section 4.7.

3 Constants

A constant is a global immutable (mutable in current version though) variable, the value of which is known at compile time:

```
let my_name: str = "George";
let ball_radius: f32 = 50;
let cyan: color = color {
  r: 0,
  g: 255,
  b: 255,
  a: 255
};
```

In the current version the type of a constant should be explicitly specified.

4 Types

Every Good expression has a type. A type indicates what kind of value can the expression possibly evaluate to. For example `2 + 2 * 2` might have a type of `i32`, while `"Hello, world"` will be of type `str`. There are primary and compound types.

4.1 Primary Types

Primary types represent low-level data such as integers, booleans and floating-point numbers. Among primary types are

- signed integer types
  - `i8` — a type of signed 8-bit integers
  - `i16` — a type of signed 16-bit integers
  - `i32` — a type of signed 32-bit integers
  - `i64` — a type of signed 64-bit integers
- unsigned integer types
  - `u8` — a type of unsigned 8-bit integers
  - `u16` — a type of unsigned 16-bit integers
  - `u32` — a type of unsigned 32-bit integers
  - `u64` — a type of unsigned 64-bit integers
- floating-point number types
  - `f32` — a type of 32-bit floating-point numbers
  - `f64` — a type of unsigned 64-bit floating-point numbers
- `bool` — a type of boolean values, `true` and `false` being its only members
- `()` — unit type, similar to C's `void`. Its only member is the unit `()`

4.2 Structs

To define a type that represents a pack of multiple related values one can use structs:

```
struct prisoner {
  height: f32,
  id: u32,
  name: str
}
```

A member of such type can be created using the following syntax:

```
fn imprison_me() prisoner
  do prisoner {
    height: 1.81,
    id: 998244323,
    name: "George"
  }
```

If there is an item with the name of a field in the current scope, a short form can be used:

```
fn imprison_me_shortly() prisoner {
  let height = 1.81;
  let id = 998244353;
  let name = "George";
  prisoner { height, id, name }
}
```

Each field of a struct can be accessed with the following syntax:

```
fn get_name(prisoner) str
  do prisoner.name
fn get_id(prisoner) u32
  do prisoner.id
```

When declaring a struct, the name of a field can be inferred from its type:

```
struct i32_and_f32 { i32, f32 }
struct str { *u8, size }
```

The first struct has two fields: `i32` and `f32`, the second one has `ptr` and `size`. More on type-to-name inference in Section 4.7.

4.3 Enums

To define a type by enumerating its possible variants one can use enums:

```
enum traffic_light {
  red,
  yellow,
  green
}
```

A member of such type then can be created using one of the variants:

```
fn current_light() traffic_light
  do if time_to_go()
    do green
  else red
```

One might want to change the control flow depending on the variant of an enum. One of the solutions is to use if-exspressions and comparisons:

```
fn driver_logic() {
  let light = current_light();
  if light == red
    do wait_impatiently()
  else if light == yellow
    do prepare()
  else if light == green // or just `else`
    do start_driving()
}
```

But there is a better way to do it, which is pattern matching:

```
fn better_driver_logic()
  do current_light().match {
    red    => wait_impatiently(),
    yellow => prepare(),
    green  => start_driving()
  }
```

Enum variant can also carry a value of a different type:

```
enum command_result {
  success,
  fail(i32)
}
```

Creating a variant with a value is identical to calling a function:

```
fn run_command() command_result
  do os().match {
    linux => success,
    windows => fail(666)
  }
```

When pattern matching on such enums, a value can be used with the following syntax:

```
fn just_do_it()
  do loop launch_missiles().match {
    success => {
      log_info("missiles launched");
      break;
    },
    fail(code) => {
      log_error("failed to launch missiles, error code: {code}");
      log_info("trying again");
    }
  }
```

4.4 Type Aliases

A type alias is a way to give a new name for an existing type:

```
type name = str;
```

Now you can use `name` type that will automatically be resolved into `string`:

```
fn do_nothing(str) name do str

fn my_name() name do "george"

fn main()
  do my_name()
    .do_nothing()
    .println()
```

4.5 Pointers

Good language provides a reliable and performant memory management model, but sometimes a programmer still needs to take the control over memory in their hands. Pointer types exist exactly for such situations. Consider for example a possible implementation of the `vec` type (akin to C++‘s `vector`):

```
struct vec<t> {
  ptr: *t,
  len: u64,
  cap: u64
}
```

Here the `ptr` field has the type `*t` that can be read as “a pointer to a member of type `t`”. This type represents a number that, in turn, represents a place in a computer’s memory where the value of type `t` is stored.

In C language such pointers are oftenly used to represent an array of values of type `t`. In order for such use case to be possible in Good programs, when adding a number to a pointer, the former is multiplied by the byte size of the `t` value (By the way, the size of a type can be aquired in the code with `@sizeof your_type`).

To point to an existing item, one can use the following syntax:

```
let my_name: str = "George";

fn my_name_memory_location() *str
  do &my_name
```

There is also a dot-notation for this operator:

```
fn dotted_mnml() *str
  do my_name.&
```

It is, however, rarely used, as it looks kinda ugly.

A pointer can be dereferenced:

```
fn first_char(vec<u8>) u8
  do *vec.ptr
```

There is also a dot-notation for this operator:

```
fn dotted_first_char(vec<u8>) u8
  do vec.ptr.*
```

Also, like in C, a pointer can be indexed:

```
fn nth_char(vec<u8>, n: u64) u8
  do vec.ptr[u64]
```

The expression above is equivalent to `*(vec.ptr + u64)`.

4.6 Function types

The Good function is also a value, and thus it has its own type:

```
fn add(a: i32, b: i32) i32
  do a + b

let add_const: fn(i32, i32) i32 = add;

fn main()
  do add_const(123, 321).println()
```

Here the type of `add` and `add_const` is `fn(i32, i32) i32`, and the type of `main` is `fn()`. Notice that the types `fn(a, b) ()` and `fn(a, b)` are equivalent, just like the definitions `fn f(a, b) () {}` and `fn f(a, b) {}` are.

4.7 Type-to-name inference

The names are inferred from the types as follows:

Given type	**Inferred name**
`some_name`	`some_name`
`*_`	`ptr`
`&<type>`	name of `<type>`
`fn( … ) _`	`fn`

`[_]`	`arr`

4.8 Arrays sugar syntax

The `arr<t>` type is widely used across Good programs, and some of the language constructs produce values of this type. That is why it was given a special syntax: `[t]`.

5 Generics

A thing is said to be “generic” if it can be used with an arbitrary type. Good programs can have generic types and generic functions. An example of a generic type is the Good array type:

```
struct arr<t> { *t, size }
```

Now to make a type of, for example, arrays of strings, one should write `arr<str>` (or `[str]` in case of arrays). In fact, the `str` type itself is defined as

```
type str = [u8]; // = arr<u8>;
```

When constructing such a type, the generic argument is inferred automatically:

```
fn main() {
  let my_raw_name = "George"; // `my_raw_name` has type `str`
  let all_my_raw_names = arr {
    ptr: &my_raw_name, // `&my_raw_name` has type `*str`
    size: 1
  }; // `arr` has type `[str]`
}
```

But what if one wants to make a function that works with values of generic types? In order to do so, there are generic functions:

```
fn byte_size<t>([t]) u64
  do arr.size * @sizeof t

fn first<t>([t]) t
  do *arr.ptr
```

I really think it is pretty clear how they work at this point. The generic arguments to such functions are also inferred automatically:

```
fn main() {
  let my_name = "George"; // `my_name` has type `str` = `[u8]`
  let my_initial = my_name.first(); // `my_initial` has type `u8`
  println("I am {my_initial}. Kozirev")
}
```

There only remains to be said that both types and functions can have more than one generic parameter. I do believe that the reader will manage to deduce how to do so.

6 Traits

In many programming languages there is a function `to_string` that converts data to string format. Let's define such function:

```
fn to_string(cat) string
  do "cat {cat.name}" as string // cast from `format_str` to `string`

fn to_string(human) string
  do human.name
```

If you try to run this code (having `cat` and `human` properly defined), you will get an error:

```
! error checking test.good at 12:4:
     |
  12 | fn to_string(human) str
     |    ``````````
--! item `to_string` is already declared in test.good at 9:4:
     |
   9 | fn to_string(cat) str
     |    ``````````


! check failed with 1 error
```

The problem is clear: we need a way to define a function with different behaviour for different types. That is where traits are used:

```
trait display {
  fn to_string(self) string;
}

impl display for cat {
  fn to_string(self) string
    do "cat {self.name}" as string
}

impl display for human {
  fn to_string(self) string
    do self.name
}
```

Here `self` is used to refer to a type for which the trait is implemented. In general, the syntax is as follows:

```
trait trait_name {
  fn f1();
  fn f2();
   …
}

impl trait_name for <type> {
  fn f1() {
     …
  }

  fn f1() {
     …
  }
   …
}
```

Another common case is when a programmer wants to define a function for all types with some methods. For example, we want to have `println` for each type that has `to_string` (i.e. that implements `display`). We can do so by **constraining** the generic parameters:

```
fn println<t: display>(t)
  do t.to_string().puts()
```

This function can be used only with those `t` that implement `display` and hence can be passed to `to_string` function:

```
impl display for human { … }

fn main()
  do my_cat.println()
```

results in

```
! error checking test.good at 30:6:
     |
  30 |    do my_cat.println()
     |       ~~~~~~~
--! type cat does not implement trait `display`
--@ the trait is implemented by:
    - human


! check failed with 1 error
```

7 C Compatibility

The Good language is fully compatible with C. What that means is that any C library can be imported in Good programs, and vice verca, any Good library can be used in C programs[1].

The former can be done using the following syntax:

```
extern fn sqrt(f64) f64;
extern fn realloc<t>(*t, size) *t;
```

This states that these functions will be in scope during the linkinga fase of the compiler. If it turns out not to be true, the linker will throw an error:

```
! error running `cc`:
─── stderr ─────────────────────────────────────
/usr/bin/ld: /tmp/ccw6Twtf.o: in function `main':
…/out.s:6:(.text+0×5): undefined reference to `reallok'
collect2: error: ld returned 1 exit status
────────────────────────────────────────────────
```

If the type of an extern function does not correspond to its actual type, the behaviour of the function is undefined.

8 Type Inference

It could have been seen above that, when declaring a variable, one does not need to specify its type:

```
fn f(i32) {}
fn main() {
  let a = 123; // type is inferred to be `i32`
  f(a);
}
```

It can be shown that in a program that compiles without errors, the type of every variable can be inferred automatically and the explicit annotation of the variable's type is always optional. It can still be considered sometimes to clarify the code.

9 Ownership

Ownership is a memory management model used in Good language. It comes down to two rules:

[1]In fact, not *any* Good library can be used in C. That is because C does not support generics, while Good does. But functions and types without generics can indeed be used in C code.

- every **value** has unique **owner** — a variable to which it was assigned.
- when the **owner** comes out of the scope, the the **value** is **dropped**.

The process of dropping the value highly depends on the value's type:
- If it is a struct, all of its fields are dropped.
- If it is an enum variant that carries a value of different type, this value is dropped.
- In addition, if the type implements `drop` trait, its `drop` function is called on the value. With this function types like `vec<t>` can `free` the memory they allocated before.
- If none of the above is true, nothing happens.

The ownership can be passed from one **owner** to another:

```
fn main() {
  let a = [123, 321, 123] as vec; // `a` owns the vector
  let b = a; // now `b` owns the vector
  // `f(a)` here would be illegal, as `a` was moved to `b`.
  f(b) // b is moved
}

fn f(vec<i32>) {
   ...
} // here `drop(&vec)` is called

impl<t> drop for vec<t> {
  fn drop(&self)
    do free(self.ptr)
}
```

Here you can see that `drop` recieves `&vec`, not just `vec`. That is a reference type. It is used to access the values without recieving the ownership:

```
fn byte_size<t>(&vec<t>) size
  do vec.size * @sizeof t

fn main() {
  let a = [123, 321] as vec;
  byte_size(&a).println()
  // `a` still owns the vector here
  a.first().println()
}
```

But is it not always good to take the reference, not the ownership? No, it is not. There are two cases when you would prefer to pass the ownership to a function:
- The value will be modified and should not be used after the end of the function. Then it is better to move the value, so that it cannot be used after.
- The value will never be used after the end of the function and it is more memory-efficient to call `free` in the function than to do it after.

The latter however should very rarely be the reason because "never" there is a huge assertion that almost always turns out to be false.

10 Coercion

It is said that a value of type `a` **coerces** to a value of type `b` if wherever the latter is expected, the former is also accepted.

To define a coercion between types `a` and `b`, a `coerce` trait should be implemented:

```
trait coerce<to> {
  fn coerce(self) to;
}

impl coerce<cat> lion { … }

fn f(cat) { … }

fn main() {
  let a: lion = … ;
  f(a);
}
```

11 Error System

There are two ways of handling errors in the Good language: `res` enum way, which I will call `?-way` and the Good error system way, which I will call `!-way`. These approaches can also be combined into `!?-way`, which is (imho) rarely useful and into `?!-way`, which can (imho) be useful more often.

11.1 Doing errors ?-way

The first way is to return a `res<t, error>` instead of just `t` in an error-ish function, where `res` is defined as

```
enum res<r, e> {
  ok(r),
  err(e)
}
```

With this way working with errors is the same as working with any other value:

```
fn get_present(boy) res<present, trash>
  do boy.attitude.match {
    good_boy ⇒ from_santa().ok(),
    bad_boy ⇒ err(coal)
  }

fn main() {
  ma_boi.get_present().match {
    ok(present) ⇒ println("yay! I've got {present}!"),
    err(trash) ⇒ println("Oh nein! Ich habe {trash} bekommen!")
  }
}
```

But that is not *the right way* dictated by the local tyranny (by me), so there is a second way, **the right way**.

11.2 Doing errors !-way

```
enum punishment {
  shoot(head),
  no_vodka
}

fn get_present(comrade) present !punishment
  do if comrade.work_hours > quota_5_years
    do [vodka, ushanka] as present
  else throw shoot(comrade.head)

fn main() {
  comrade.get_present().send_to_family().catch punishment {
    let corpse = execute(punishment);
    bury(corpse);
  }
}
```

Such errors automatically fall through if not catched:

```
fn f() i32!error
  do if is_dooms_day()
    do throw most_fatal_error
  else 2

fn g() i32!error
  do 2 + f() * 2
```

If an error-ish function is used in a normal one, the error must be catched:

```
fn h() i32
  do (2 + g() * 2).catch err do die(err)
```

11.3 Combining into `!?+?!-way`

The first-way functions can easily be converted to the second-way with

```
fn second_way() i32 !error
  do first_way()!
```

And the second-way functions can easily be converted to the first-way with

```
fn first_way() res<i32, error>
  do second_way()?
```

But how to decide when to use one way or another? The answer is — practically doesn't matter: they are so easy to convert from one to another and are compiled into same machine code. The language aims to allow any style of programming and the single rule is to use whatever you personally prefer. It is good to know though that it is much easier to chain error-ish actions in the second mode and you have better grip over control flow in the first.

Anyway, here is a table comparing approaches on standard error-related operations:

?-way	**!?-way**	**!-way**
`f().map(|x| g(x))`	`f()!.g()?`	`f().g()`
`f().then(|x| g(x))`	`f()!.g()!?`	`f().g()`
`f().ok_or(|x| g(x))`	`f()!.catch x do g(x)`	`f().catch x do g(x)`
`f().or(|x| g(x))`	`f()!.ok().catch x do g(x)`	`f().catch x do g(x)`
`f().map_err(|x| g(x))`	`f()!.ok().catch x do err(g(x))`	`f().catch x do throw g(x)`
long pattern match	`f()!.catch x do return err(x)`	`f()`

12 Impl derivation

work in progress

13 Expressions

work in progress

14 Syntax sugar

work in progress