FROM IMPERATIVE TO FUNCTIONAL APIS

Marek Kubica @leonidasfromxiv

28. March 2015

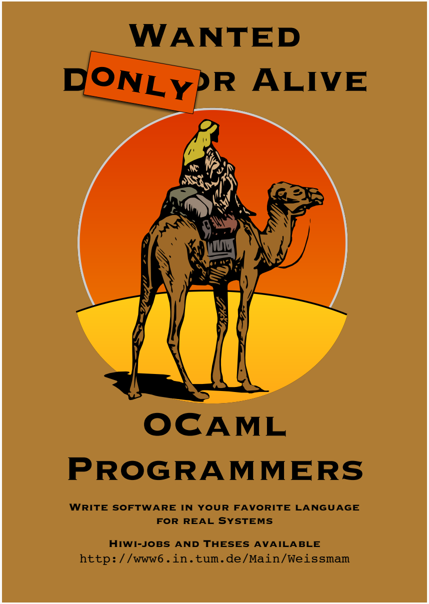
A WORD FOR THE SPONSORS

stλlefruits

- They sponsored me going to this conference
- Jan Stępień is holding a talk in the currying hall at 11:00
- … and a workshop at 14:00!
- Thanks, stylefruits!

WHO AM I?

- Marek Kubica
- Student at the TUM
- I do free software
- Dabbled in just about every language evar


Wanted
D
Only
or Alive
OCaml Programmers
Write software in your favorite language for real Systems
Hiwi-jobs and Theses available
http://www6.in.tum.de/Main/Weissmam

SO, WHY?

- Data compression support in OCaml is… kinda meh
- Using OCaml foreign function interface to talk to libarchive
- Make it a bachelor thesis!
- Thought I might as well create a better API
- What *IS* a better API anyway?

SO, WHY?

- Data compression support in OCaml is... kinda meh
- Using OCaml foreign function interface to talk to libarchive
- Make it a bachelor thesis!
- Thought I might as well create a better API
- What *IS* a better API anyway?

My goal for today

Show you that advanced static type features are not (only) academic.

STATIC TYPING THE C WAY

Let’s check how C handles look like. How does libarchive handle this?

$$\underbrace{\texttt{\_\_LA\_DECT struct archive*}}_{\text{C return type}}\underbrace{\mathit{archive\_read\_new}}_{\text{Function name}}(\quad\underbrace{\mathit{void}}_{\text{Argument type}}\quad);$$

$$\underbrace{\texttt{\_\_LA\_DECT struct archive*}}_{\text{C return type}}\underbrace{\mathit{archive\_write\_new}}_{\text{Function name}}(\quad\underbrace{\mathit{void}}_{\text{Argument type}}\quad);$$

- Opaque pointer to some struct
- Write handles and read handles have the same type

So, what can we do with these handles?

So, what can we do with these handles?

- Create them
- Open them
- Configure them
- Read from them
- Write to them
- Close them

So, what can we do with these handles?

- Create them
- Open them
- Configure them
- Read from them
- Write to them
- Close them

Cool.

So, what can we do with these handles?

- Create them
- Open them
- Configure them
- Read from them
- Write to them
- Close them

Cool. But what if we screw up?

SEGFAULT

```
zsh: segmentation fault (core dumped)  ./errors
```

DOUBLE FREE

```
*** Error in ‘./errors’: double free or corruption (fasttop): 0x000000000077a010 ***
======= Backtrace: =========
/usr/lib/libc.so.6(+0x788ae)[0x7fa0c97cd8ae]
/usr/lib/libc.so.6(+0x79587)[0x7fa0c97ce587]
./errors[0x40057b]
/usr/lib/libc.so.6(__libc_start_main+0xf5)[0x7fa0c9776a15]
./errors[0x400479]
======= Memory map: ========
00400000-00401000 r-xp 00000000 fe:01 21244012                         /lambdacon/errors
00600000-00601000 rw-p 00000000 fe:01 21244012                         /lambdacon/errors
0077a000-0079b000 rw-p 00000000 00:00 0                                [heap]
7fa0c953f000-7fa0c9554000 r-xp 00000000 fe:00 4213606          /usr/lib/libgcc_s.so.1
7fa0c9554000-7fa0c9754000 ——-p 00015000 fe:00 4213606          /usr/lib/libgcc_s.so.1
7fa0c9754000-7fa0c9755000 rw-p 00015000 fe:00 4213606          /usr/lib/libgcc_s.so.1
7fa0c9755000-7fa0c98f8000 r-xp 00000000 fe:00 4202529          /usr/lib/libc-2.17.so
7fa0c98f8000-7fa0c9af8000 ——-p 001a3000 fe:00 4202529          /usr/lib/libc-2.17.so
7fa0c9af8000-7fa0c9afc000 r——p 001a3000 fe:00 4202529          /usr/lib/libc-2.17.so
7fa0c9afc000-7fa0c9afe000 rw-p 001a7000 fe:00 4202529          /usr/lib/libc-2.17.so
7fa0c9afe000-7fa0c9b02000 rw-p 00000000 00:00 0
7fa0c9b02000-7fa0c9b23000 r-xp 00000000 fe:00 4203728          /usr/lib/ld-2.17.so
7fa0c9cfa000-7fa0c9cfd000 rw-p 00000000 00:00 0
7fa0c9d22000-7fa0c9d23000 rw-p 00000000 00:00 0
7fa0c9d23000-7fa0c9d24000 r——p 00021000 fe:00 4203728          /usr/lib/ld-2.17.so
7fa0c9d24000-7fa0c9d25000 rw-p 00022000 fe:00 4203728          /usr/lib/ld-2.17.so
7fa0c9d25000-7fa0c9d26000 rw-p 00000000 00:00 0
7fff44461000-7fff44482000 rw-p 00000000 00:00 0                [stack]
7fff445fe000-7fff44600000 r-xp 00000000 00:00 0                [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0        [vsyscall]
zsh: abort (core dumped)  ./errors foo
```

What actually happens: libarchive returns `ARCHIVE_FATAL`.

What actually happens: libarchive returns `ARCHIVE_FATAL`.

Unless you trigger a bug in libarchive.

What actually happens: libarchive returns `ARCHIVE_FATAL`.

Unless you trigger a bug in libarchive. Then it segfaults.

LOTS OF THINGS CAN GO WRONG

- Reading from handle that is not open *
- Writing to a read handle
- Not setting the options correctly (compression formats)

LOTS OF THINGS CAN GO WRONG

- Reading from handle that is not open *
- Writing to a read handle
- Not setting the options correctly (compression formats)

* this actually happened

NOT TO GRIPE ON LIBARCHIVE

Public Service Announcement

libarchive is a rather well designed library. Mostly idiomatic C, so don't think this is a deliberately bad example. It is how things *are* in C land.

- This is an OK API for C.
- Fragile APIs are common in C.

NOT TO GRIPE ON LIBARCHIVE

Public Service Announcement

libarchive is a rather well designed library. Mostly idiomatic C, so don't think this is a deliberately bad example. It is how things *are* in C land.

- This is an OK API for C.
- Fragile APIs are common in C.

But can we do better?

Yes

Yes

(obviously)

FIX UP THE HANDLE TYPES

How to prevent writing to read handles and reading from write handles?

```
external read_new: unit -> archive = ”ost_read_new”
external write_new: unit -> archive = ”ost_write_new”
```

FIX UP THE HANDLE TYPES

How to prevent writing to read handles and reading from write handles?

```
external read_new: unit -> archive = ”ost_read_new”
external write_new: unit -> archive = ”ost_write_new”
```

Yup, create different handle types.

```
type r = archive
type w = (archive * write_buffer_ptr * written_ptr)
```

So now we have distinct types to represent handles.

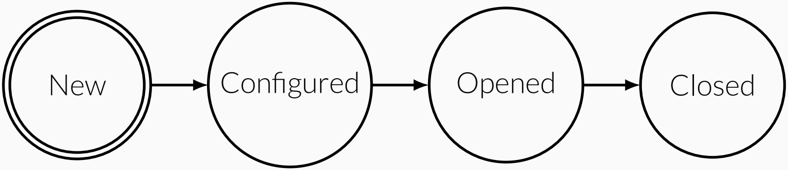
ACHIEVEMENT UNLOCKED!

Type safety improved
Writing read handles disallowed

But of course you aren't attending the talk for this trivial epiphany. We can do this easily in C as well! Let's do better.

OUR HANDLES HAVE STATES!

The handles always traverse some fixed states:



Couldn’t we encode the state in the type somehow?

ADDING STATE INFORMATION IN THE TYPE

We can add state. OCaml has parametrized types *:

```
# [];;
- : ’a list = []
# [1];;
- : int list = [1]
# type ’a read_handle = ReadHandle of ’a;;
type ’a read_handle = ReadHandle of ’a
```

* if you haven’t seen them, think of them kinda like generics

ASIDE: OPEN UNION TYPES

So, now we can parametrize types with other types.

We could create our own state types:

```
type state = New | Configured | Opened | Closed
```

ASIDE: OPEN UNION TYPES

So, now we can parametrize types with other types.

We could create our own state types:

```
type state = New | Configured | Opened | Closed
```

But these can’t be extended if someone wants to add a new state.

ASIDE: OPEN UNION TYPES

So, now we can parametrize types with other types.

We could create our own state types:

```
type state = New | Configured | Opened | Closed
```

But these can't be extended if someone wants to add a new state.

Plus, we're lazy. Let's use open union types aka polymorphic variants:

```
[‘New] [‘Configured] [‘Opened] [‘Closed]
```

Great, so now we can create functions that *require* handles of the correct state.

e.g. a `read` function that only works on `[‘Opened] read_handle`.

Great, so now we can create functions that *require* handles of the correct state.

e.g. a `read` function that only works on `[‘Opened] read_handle`.

Foiled again!

Great, so now we can create functions that *require* handles of the correct state.

e.g. a `read` function that only works on `[‘Opened] read_handle`.

Foiled again!

The OCaml compiler is too smart, it knows that `[‘Opened] read_handle` is the same type as `[‘New] read_handle`. Therefore every function which takes the `[‘Opened]` handle accepts every other type of handle as well.

ENTER PHANTOMS

We'd need to hide the actual `read_handle` type from the compiler.

ENTER PHANTOMS

We'd need to hide the actual `read_handle` type from the compiler.

Boy oh boy, we can!

ENTER PHANTOMS

We’d need to hide the actual `read_handle` type from the compiler.

Boy oh boy, we can!

We create a module and only say:

```
module Handle : sig
  type ’a r
  (* our signatures *)
  val new : unit -> [‘New] r
end = struct
  type ’a r = read_handle
  (* our functions *)
  external new : unit -> [‘Open] r = ”ost_read_new”
end
```

ACHIEVEMENT UNLOCKED!

Type safety improved
Made API misuse a type error

✓Writing read handles disallowed

✓Using the proper handle in an incorrect way disallowed

And now for something completely different!

HAVE YOU EVER SEEN THIS?

Ever used Python?

```
Traceback (most recent call last):
File ”<stdin>”, line 1, in <module>
AttributeError: ’NoneType’ object has no attribute ’foo’
```

HAVE YOU EVER SEEN THIS?

Ever used Python?

```
Traceback (most recent call last):
File ”<stdin>”, line 1, in <module>
AttributeError: ’NoneType’ object has no attribute ’foo’
```

Ever touched Java?

```
Exception in thread ”main” java.lang.NullPointerException
at NPE.main(NPE.java:8)
```

HAVE YOU EVER SEEN THIS?

Ever used Python?

```
Traceback (most recent call last):
File ”<stdin>”, line 1, in <module>
AttributeError: ’NoneType’ object has no attribute ’foo’
```

Ever touched Java?

```
Exception in thread ”main” java.lang.NullPointerException
at NPE.main(NPE.java:8)
```

Ever seen C?

```
zsh: segmentation fault (core dumped)  ./errors
```

YOU KNOW WHOSE FAULT IT IS!

- `null`
- `None`
- `NULL`

YOU KNOW WHOSE FAULT IT IS!

- `null`
- `None`
- `NULL`

Everytime you return null as a placeholder value, ~~$DEITY kills a kitten~~ you have to check whether you weren't handed null in return.

Let’s kill the ~~Batman~~ Null pointer!

ATTEMPT ONE: EXCEPTIONS

- Common solution
- Ubiquitous (Java, Python, C++, Ruby, whathaveyou)
- Easy to understand
- OCaml does have exceptions

ATTEMPT ONE: EXCEPTIONS

- Common solution
- Ubiquitous (Java, Python, C++, Ruby, whathaveyou)
- Easy to understand
- OCaml does have exceptions
- Not typesafe, unless you consider checked exceptions
- Boring!

ATTEMPT TWO: OPTION TYPES

Observation: everythime we return NULL, we either return something meaningful or an invalid placeholder.

We might even say:

```
type 'a option = Some of 'a | None
```

ATTEMPT TWO: OPTION TYPES

Observation: everythime we return NULL, we either return something meaningful or an invalid placeholder.

We might even say:

```
type ’a option = Some of ’a | None
```

Therefore, everytime a function returns `’a option` we have to pattern match:

```
let optional x = Some x
match optional 42 with
  | Some x -> x
  | None -> 0
```

ATTEMPT TWO: OPTION TYPES

Observation: everythime we return NULL, we either return something meaningful or an invalid placeholder.

We might even say:

```
type 'a option = Some of 'a | None
```

Therefore, everytime a function returns `'a option` we have to pattern match:

```
let optional x = Some x
match optional 42 with
  | Some x -> x
  | None -> 0
```

If we forget:

ACHIEVEMENT UNLOCKED!

Type safety improved
Missed null check is type error

✓No more Null pointer failures on runtime!

Everything is fun and games until you need to specify a reason for failure.

What if we could add an error message?

```
type ('a, 'b) err = Success of 'a | Failure of 'b
```

Done!

But pattern matching on every function call sucks because it is tedious! Just look at this mess:

```
match firstfn 42 with
  | Success (x) -> (match secondfn x with
    | Success (y) -> (match thirdfn y with
      | Success (z) -> z
      | Failure (f3) -> ”Failure at thirdfn”)
    | Failure (f2) -> ”Failure at secondfn”)
  | Failure (f1) -> ”Failure at fristfn”
```

Right. Maybe we can simplify...

In Haskell, `option` is called “Maybe monad” and `error` is called “Error monad”.

But pattern matching on every function call sucks because it is tedious! Just look at this mess:

```
match firstfn 42 with
  | Success (x) -> (match secondfn x with
    | Success (y) -> (match thirdfn y with
      | Success (z) -> z
      | Failure (f3) -> "Failure at thirdfn")
    | Failure (f2) -> "Failure at secondfn")
  | Failure (f1) -> "Failure at fristfn"
```

Right. Maybe we can simplify...

In Haskell, `option` is called “Maybe monad” and `error` is called “Error monad”.

BAM, SCARY MONADS!

Haskell features an operator called `bind` aka `»=` to chain operations on monads.

```
val bind: ’a ErrorMonad.t ->
  (’a -> ’b ErrorMonad.t) ->
  ’b ErrorMonad.t
```

`bind` takes an error monad wrapping type `’a`, and a function which takes `’a` and returns an error monad wrapping `’b` and returns that value.

Basically an unwrapper function.

For the error monad, it looks like this:

```
let bind m f = match m with
  | Success(x) -> f x
  | Failure(f) -> Failure(f)
```

We can use it like this:

```
match (bind (bind (firstfn 42) secondfn) thirdfn)
with
  | Success (x) -> x
  | Failure (_) -> ”Failure in chain”
```

The code got a lot easier!

ASIDE: OPERATOR TRICKS

OCaml allows custom operators as long as they follow naming rules.

```
let (»=) = bind
```

Using it is easy:

```
match (firstfn 42) »= secondfn »= thirdfn with
  | Success (x) -> x
  | Failure (_) -> ”Failure in chain”
```

ACHIEVEMENT UNLOCKED!

✓No more Null pointer failures on runtime!

✓Easy and convenient to get reason of failure

HOW TO CONTINUE FROM HERE

These are good steps but we don’t have to stop here:

- Making invalid state unrepresentable in the type system
- Generalized Algebraic Data Types allow additional restrictions
- Use polymorphic variants as markers for different types of error (`‘File_not_found`, `‘Permission_denied`)

Take care: the API might turn out to be *too complicated*. Please, use common sense*.

HOW TO CONTINUE FROM HERE

These are good steps but we don't have to stop here:

- Making invalid state unrepresentable in the type system
- Generalized Algebraic Data Types allow additional restrictions
- Use polymorphic variants as markers for different types of error (`‘File_not_found`, `‘Permission_denied`)

Take care: the API might turn out to be *too complicated*. Please, use common sense*.

* if not applicable, emulate idioms from good APIs in your preferred programming language.

Marek Kubica

Check out my playthings:

- Leonidas-from-XIV on GitHub
- @leonidasfromxiv
- `https://xivilization.net/`