Purely functional programming

There is no turning back

Marek Kidoň

February 23, 2017

What is purely functional programming

Functional

- Style of building the structure of computer programs.
- Declarative: The job is done using expressions(declarations), not statements.
- Function output depends on inputs only.

Purely functional

- Controversy: A lot of things to a lot of people.
- How do you handle computational side effects purely?
- Treatment of computational effects is the key.

Why do we like it

- Very expressive
- Highly polymorphic
- Parallelism and concurrency
- Strong static typing

Types: Motivation

- Mars Climate Orbiter disintegrated during trajectory correction in 1999.
- Cause: Software provided by Lockheed Martin calculated in pound-seconds. NASA expected newton-seconds. Cost: $125 million.
- Ariane 5 was destructed by engineers shortly after launch.
- Cause: Computers tried to cram 64bit number into 16bit space. Cost: $500 million worth of the carried satellite.
- Some dude's website kicked a bucket.
- Cause: An obscure function expected a capitalized string but received a lowercase one. Cost: a really bad day for some dude and his employer.
- Every day a lot of money is wasted on bugs, their consequences and their fixing (which often introduces new bugs...).

Types: What are they?

- Types and static type systems are programmers best friend!
- Controversy: a lot of things to a lot of people.
- Definition: Syntactic? Representational? Behavioral? Value space definition?
- Common sense: Type is uniquely defined by the set of its values. Think of integers, characters, tuples, lists, etc...

```
Word32 = 0 | 1 | 2 | 3 | ... | 4294967295
SymbolTuple =  AA ('a','a') | ... | ZZ ('z','z')
```

Types: The epic fail...

- How do you define a new type in most programming languages?
- How do you represent the Logic = {True, False, Unknown, Uninitialized, ...} type used in hardware related applications?
- Are data structures or objects the solution?
- Answer: It depends...
- Common practice is to map type semantics into a structure of existing types. Are their sets of values isomorphic?

Types: Example 1

```
#define TRUE 0
#define FALSE 1
#define UNKNOWN 2
#define UNINITIALIZED 3
typedef unsigned int STD_LOGIC;
void launchViciousRocket(STD_LOGIC);

int main(void) {
        STD_LOGIC signalValue;

        signalValue = UNKNOWN;
        signalValue = 0xdeadbeef;

        launchViciousRocket(signalValue);

        return 0;
}
```

- Question: Will the vicious be rocket launched?
- Answer: Who knows...

Types: Example 2

```
void launchViciousRocket(STD_LOGIC signal) {
    switch(expression) {
        case TRUE :
            killMillionsOfPeople();
            break;

        case FALSE :
            doNotKillMillionsofPeople();
            break;

        case UNKNOWN :
            newSignal = flipACoin();
            launchViciousRocket(newSignal);
            break;
    }
    restOfTheStatements();
}
```

- Question: will be the vicious rocket launched?
- Answer: Who knows...

Types: The purely functional approach

- You can create new types as labels for their sets of values.
- A value of a certain type is instantiated by picking a member from its set.
- Pattern matching: as opposed to creating, you have a way to dismantle it.
- If you do not match on all members of the type compiler may complain.

Types: Example 3

```
launchViciousRocket
    :: StdLogic
    -> PossiblyKillMillionsOfPeople

data StdLogic
    = TRUE
    | FALSE
    | Unknown
    | Uninitialized

main =
    let signalValue = 0xdeadbeef
    in launchViciousRocket(signalValue)
```

- Question: will be the vicious rocket launched?
- Answer: No, such program will never compile

Types: Example 4

```
launchViciousRocket
    :: StdLogic
    -> PossiblyKillMillionsOfPeople

launchViciousRocket signal = case signal of
    TRUE -> killMillionsOfPeople
    FALSE -> doNotKillMillionsofPeople
    Unknown -> withFlipACoin launchViciousRocket
    // Uninitialized ->
```

- Question: Will be the vicious rocket launched?
- Answer: Who knows...

Types: The lesson

- Computer program semantics can be implemented in types.
- Type correctness can be checked at compile time!
- Sadly you can't create new types in most programming languages.
- You rely on run-time checks instead of compile-time type checker.

The Vision case study

Task

- Write an automated quality control system for Hyundai.
- You have less than 2 months for it.
- Two people project.
- Customer has to sign an acceptation protocol.

Result

- 10000 loc of Haskell
- Few hundreds lines of C++/openCV using functional techniques.
- Not a single unit test.
- Some functional tests.
- Acceptation protocol signed.
- Still waiting for first bug report.

PFP: Programmer's mindset

- Problem: People do make mistakes.
- Solution: Computers do not.
- Problem: People hate writing boilerplate code. It is repetitious, boring and error prone.
- Solution: Use as many abstract constructs as possible. Do not write boilerplate code. Good compiler can generate it for you.
- Problem?: People are lazy.
- Solution: Be lazy to type. *Do not be lazy to think*. Encode semantics in types whenever it is beneficial. Let the compiler do your work.

PFP: What does it bring to a company

- Imagine you will do the next project in Haskell...
- What are the consequences?
- You get smart and creative people by definition(previous slide).
- What is their motivation?
- Your software is more likely to be less buggy

PFP: There is no turning back

- Do not fear the purely functional platforms. They are often the *final frontier* of programming techniques.
- Imagine you decide to learn purely functional programming incorporating strong static typing.
- Problem: *There is no turning back*
- Cause: You get lazy beyond belief
- Solution: Its not really a problem is it?