Definitions of Functional Programming

Comparing four Definitions

slides by @philip_schwarz

Definitions of Functional Programming

Functional programming is a way of writing software applications using only **pure functions** and **immutable values**.

A **pure function** is a function that depends only on its declared input parameters and its algorithm to produce its output. It does not read any other values from "the **outside world**" — the world outside of the **function's scope** — and it does not modify any values in the **outside world**

1. A **pure function** depends only on (a) its declared input parameters and (b) its algorithm to produce its result. A **pure function** has no "**back doors**," which means:

    1. Its result **can't depend on reading** any **hidden value** outside of the **function scope**, such as **another field in the same class** or **global variables**.
    2. It **cannot modify any hidden fields** outside of the **function scope**, such as other **mutable fields in the same class** or **global variables**.
    3. It **cannot depend on any external I/O**. It can't **rely on input** from files, databases, web services, UIs, etc; it can't **produce output**, such as **writing** to a file, database, or web service, **writing** to a screen, etc.

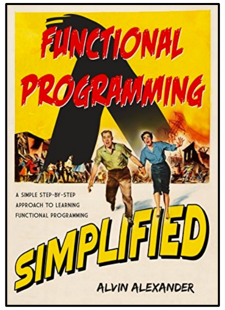
2. A **pure function does not modify** its input parameters.

This is unlike an **OOP** method, which **can depend** on other fields in the same class as the method.

As a result of 1, if a **pure function** is called with an input parameter x an infinite number of times, it will always return the same result y.

A pure function has **no side effects**, meaning that it does not read anything from the **outside world** or write anything to the **outside world**.

PF = ODI + NSE
Pure **F**unctions **=** **O**utput **D**epends only on **I**nput **+** **N**o **S**ide **E**ffects)

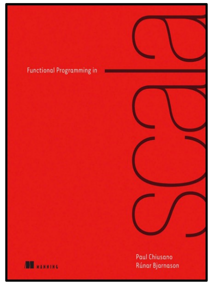


by Alvin Alexander
@alvinalexander

FP means programming with **pure functions**, and a **pure function** is one that lacks **side effects**…

A function **f** with input type **A** and output type **B** (written in Scala as a single type: **A => B** , pronounced "**A** to **B** " or "**A** arrow **B** ") is a computation that relates **every value a** of type **A** to **exactly one value b** of type **B** such that **b** is determined **solely** by the value of **a**. Any **changing state** of an internal or external process is **irrelevant to computing the result f(a)**. For example, a function intToString having type Int => String will take every integer to a corresponding string. Furthermore, **if it really is a function, it will do nothing else**.

In other words, a function has no **observable effect** on the execution of the program other than to compute a result given its inputs; we say that it has no **side effects**. We sometimes qualify such functions as **pure functions** to make this more explicit, but this is somewhat redundant.



FP in Scala
by Paul Chiusano
and Runar Bjarnason
@pchiusano @runarorama

Real AI and 14 others Retweeted

john Ⓐ De Goes @jdegoes · 30 Nov 2017

FP is **just programming** with functions. Functions are:

1. **Total:** They return an output for every input.
2. **Deterministic:** They return the same output for the same input.
3. **Pure:** Their only effect is computing the output.

The rest is **just composition** you can learn over time.

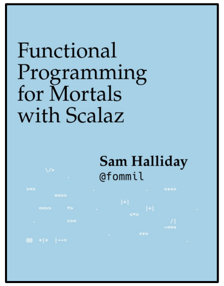
17 254 610

1.2 Pure Functional Programming

Functional Programming is the act of writing programs with **pure functions**. Pure functions have three properties:

- **Total**: return a value for every possible input
- **Deterministic**: return the same value for the same input
- **Inculpable**: no (direct) interaction with the world or program state.

Together, these properties give us an unprecedented ability to reason about our code. For example, input validation is easier to isolate with **totality**, caching is possible when functions are **deterministic**, and interacting with the world is easier to control, and test, when functions are **inculpable**. The kinds of things that break these properties are **side effects**: directly accessing or changing **mutable state** (e.g. maintaining a var in a class or using a legacy API that is **impure**), communicating with **external resources** (e.g. files or network lookup), or throwing **exceptions**.



by Sam Halliday
@fommil

Definitions of Functional Programming

Functional programming is a way of writing software applications using only **pure functions** and **immutable values**.

A **pure function** is a function that depends only on its declared input parameters and its algorithm to produce its output. It does not read any other values from "the **outside world**" — the world outside of the **function's scope** — and it does not modify any values in the **outside world**

1. A **pure function** depends only on (a) its declared input parameters and (b) its algorithm to produce its result. A **pure function** has no "**back doors**," which means:

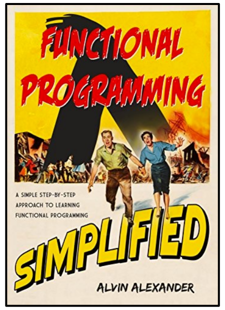
    1. Its result **can't depend on reading** any **hidden value** outside of the **function scope**, such as **another field in the same class** or **global variables**.
    2. It **cannot modify any hidden fields** outside of the **function scope**, such as other **mutable fields in the same class** or **global variables**.
    3. It **cannot depend on any external I/O**. It can't **rely on input** from files, databases, web services, UIs, etc; it can't **produce output**, such as **writing** to a file, database, or web service, **writing** to a screen, etc.

2. A **pure function does not modify** its input parameters.

This is unlike an **OOP** method, which **can depend** on other fields in the same class as the method.

As a result of 1, if a **pure function** is called with an input parameter x an infinite number of times, it will always return the same result y.

A pure function has **no side effects**, meaning that it does not read anything from the **outside world** or write anything to the **outside world**.



by Alvin Alexander
@alvinalexander

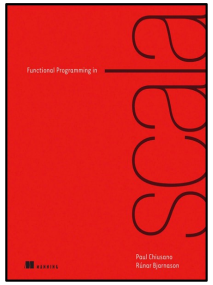
PF = ODI + NSE
Pure **F**unctions = **O**utput **D**epends only on **I**nput + **N**o **S**ide **E**ffects)

TOTALITY

FP means programming with **pure functions**, and a **pure function** is one that lacks **side effects**...

A function **f** with input type **A** and output type **B** (written in Scala as a single type: **A => B** , pronounced "**A** to **B** " or "**A** arrow **B** ") is a computation that relates **every value a** of type **A** to **exactly one value b** of type **B** such that **b** is determined solely by the value of **a**. Any changing state of an internal or external process is irrelevant to computing the result **f(a)**. For example, a function intToString having type Int => String will take every integer to a corresponding string. Furthermore, if it really is a function, it will do nothing else.

In other words, a function has no **observable effect** on the execution of the program other than to compute a result given its inputs; we say that it has no **side effects**. We sometimes qualify such functions as **pure functions** to make this more explicit, but this is somewhat redundant.



FP in Scala
by Paul Chiusano
and Runar Bjarnason
@pchiusano @runarorama

Real AI and 14 others Retweeted

john De Goes @jdegoes · 30 Nov 2017

FP is just programming with functions. Functions are:

1. Total: They return an output for every input.
2. Deterministic: They return the same output for the same input.
3. Pure: Their only effect is computing the output.

The rest is just composition you can learn over time.

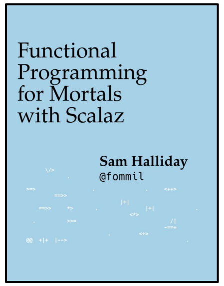
17 254 610

1.2 Pure Functional Programming

Functional Programming is the act of writing programs with **pure functions**. Pure functions have three properties:

- **Total**: return a value for every possible input
- **Deterministic**: return the same value for the same input
- **Inculpable**: no (direct) interaction with the world or program state.

Together, these properties give us an unprecedented ability to reason about our code. For example, input validation is easier to isolate with **totality**, caching is possible when functions are **deterministic**, and interacting with the world is easier to control, and test, when functions are **inculpable**. The kinds of things that break these properties are **side effects**: directly accessing or changing **mutable state** (e.g. maintaining a var in a class or using a legacy API that is **impure**), communicating with **external resources** (e.g. files or network lookup), or throwing **exceptions**.



by Sam Halliday
@fommil

Definitions of Functional Programming

Functional programming is a way of writing software applications using only **pure functions** and **immutable values**.

A **pure function** is a function that depends only on its declared input parameters and its algorithm to produce its output. It does not read any other values from "the **outside world**" — the world outside of the **function's scope** — and it does not modify any values in the **outside world**

1. A **pure function** depends only on (a) its declared input parameters and (b) its algorithm to produce its result. A **pure function** has no "**back doors**," which means:

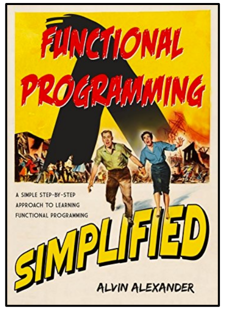
   1. Its result **can't depend on reading** any **hidden value** outside of the **function scope**, such as **another field in the same class** or **global variables**.
   2. It **cannot modify any hidden fields** outside of the **function scope**, such as other **mutable fields in the same class** or **global variables**.
   3. It **cannot depend on any external I/O**. It can't **rely on input** from files, databases, web services, UIs, etc; it can't **produce output**, such as **writing** to a file, database, or web service, **writing** to a screen, etc.

2. A **pure function does not modify** its input parameters.

This is unlike an **OOP** method, which **can depend** on other fields in the same class as the method.

As a result of 1, if a **pure function** is called with an input parameter x an infinite number of times, it will always return the same result y.

A pure function has **no side effects**, meaning that it does not read anything from the **outside world** or write anything to the **outside world**.



by Alvin Alexander
@alvinalexander

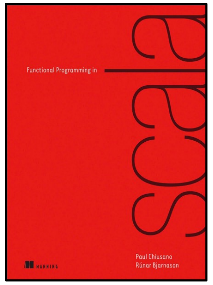
PF = ODI + NSE
Pure **F**unctions = **O**utput **D**epends only on **I**nput + **N**o **S**ide **E**ffects)

DETERMINISM

FP means programming with **pure functions**, and a **pure function** is one that lacks **side effects**…

A function **f** with input type **A** and output type **B** (written in Scala as a single type: **A => B** , pronounced "**A** to **B** " or "**A** arrow **B** ") is a computation that relates **every value a** of type **A** to **exactly one value b** of type **B** such that **b** is determined solely by the value of **a**. Any changing state of an internal or external process is irrelevant to computing the result **f(a)**. For example, a function intToString having type Int => String will take every integer to a corresponding string. Furthermore, if it really is a function, it will do nothing else.

In other words, a function has no **observable effect** on the execution of the program other than to compute a result given its inputs; we say that it has no **side effects**. We sometimes qualify such functions as **pure functions** to make this more explicit, but this is somewhat redundant.



FP in Scala
by Paul Chiusano
and Runar Bjarnason
@pchiusano @runarorama

Real AI and 14 others Retweeted

john De Goes @jdegoes · 30 Nov 2017

FP is just programming with functions. Functions are:

1. Total: They return an output for every input.
2. Deterministic: They return the same output for the same input.
3. Pure: Their only effect is computing the output.

The rest is just composition you can learn over time.

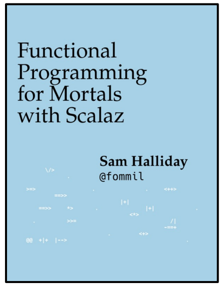
17 254 610

1.2 Pure Functional Programming

Functional Programming is the act of writing programs with **pure functions**. Pure functions have three properties:

- **Total**: return a value for every possible input
- **Deterministic**: return the same value for the same input
- **Inculpable**: no (direct) interaction with the world or program state.

Together, these properties give us an unprecedented ability to reason about our code. For example, input validation is easier to isolate with **totality**, caching is possible when functions are **deterministic**, and interacting with the world is easier to control, and test, when functions are **inculpable**. The kinds of things that break these properties are **side effects**: directly accessing or changing **mutable state** (e.g. maintaining a var in a class or using a legacy API that is **impure**), communicating with **external resources** (e.g. files or network lookup), or throwing **exceptions**.



by Sam Halliday
@fommil

Definitions of Functional Programming

Functional programming is a way of writing software applications using only **pure functions** and **immutable values**.

A **pure function** is a function that depends only on its declared input parameters and its algorithm to produce its output. It does not read any other values from "the **outside world**" — the world outside of the **function's scope** — and it does not modify any values in the **outside world**

> This is unlike an **OOP** method, which **can depend** on other fields in the same class as the method.

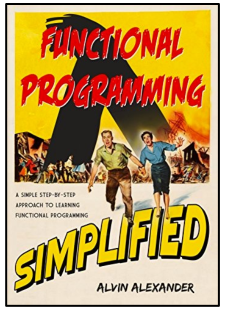
1. A **pure function** depends only on (a) its declared input parameters and (b) its algorithm to produce its result. A **pure function** has no "**back doors**," which means:

   1. Its result **can't depend on reading** any **hidden value** outside of the **function scope**, such as **another field in the same class** or **global variables**.
   2. It **cannot modify any hidden fields** outside of the **function scope**, such as other **mutable fields in the same class** or **global variables**.
   3. It **cannot depend on any external I/O**. It can't **rely on input** from files, databases, web services, UIs, etc; it can't **produce output**, such as **writing** to a file, database, or web service, **writing** to a screen, etc.

2. A **pure function** **does not modify** its input parameters.

> As a result of 1, if a **pure function** is called with an input parameter x an infinite number of times, it will always return the same result y.

> A pure function has **no side effects**, meaning that it does not read anything from the **outside world** or write anything to the **outside world**.



by Alvin Alexander
@alvinalexander

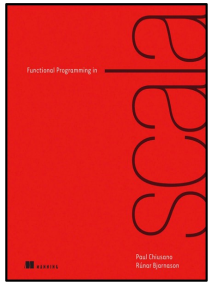
PF = ODI + NSE
Pure **F**unctions = **O**utput **D**epends only on **I**nput + **N**o **S**ide **E**ffects)

NO SIDE EFFECTS

FP means programming with **pure functions**, and a **pure function** is one that lacks **side effects**…

A function **f** with input type **A** and output type **B** (written in Scala as a single type: **A => B** , pronounced "**A** to **B** " or "**A** arrow **B** ") is a computation that relates **every value a** of type **A** to **exactly one value b** of type **B** such that **b** is determined solely by the value of **a**. Any changing state of an internal or external process is irrelevant to computing the result **f(a)**. For example, a function intToString having type Int => String will take every integer to a corresponding string. Furthermore, if it really is a function, it will do nothing else.

In other words, a function has no **observable effect** on the execution of the program other than to compute a result given its inputs; we say that it has no **side effects**. We sometimes qualify such functions as **pure functions** to make this more explicit, but this is somewhat redundant.



FP in Scala
by Paul Chiusano
and Runar Bjarnason
@pchiusano @runarorama

Real AI and 14 others Retweeted

john ⓐ De Goes @jdegoes · 30 Nov 2017

FP is just programming with functions. Functions are:

1. Total: They return an output for every input.
2. Deterministic: They return the same output for the same input.
3. Pure: Their only effect is computing the output.

The rest is just composition you can learn over time.

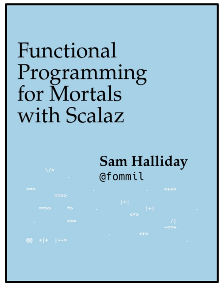
17 254 610

1.2 Pure Functional Programming

Functional Programming is the act of writing programs with **pure functions**. Pure functions have three properties:

- **Total**: return a value for every possible input
- **Deterministic**: return the same value for the same input
- **Inculpable**: no (direct) interaction with the world or program state.

Together, these properties give us an unprecedented ability to reason about our code. For example, input validation is easier to isolate with **totality**, caching is possible when functions are **deterministic**, and interacting with the world is easier to control, and test, when functions are **inculpable**. The kinds of things that break these properties are **side effects**: directly accessing or changing **mutable state** (e.g. maintaining a var in a class or using a legacy API that is **impure**), communicating with **external resources** (e.g. files or network lookup), or throwing **exceptions**.



by Sam Halliday
@fommil

Definitions of Functional Programming

Functional programming is a way of writing software applications using only **pure functions** and **immutable values**.

A **pure function** is a function that depends only on its declared input parameters and its algorithm to produce its output. It does not read any other values from "the **outside world**" — the world outside of the **function's scope** — and it does not modify any values in the **outside world**

1. A **pure function** depends only on (a) its declared input parameters and (b) its algorithm to produce its result. A **pure function** has no "**back doors**," which means:

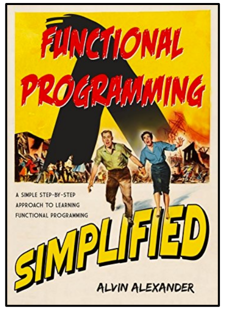
    1. Its result **can't depend on reading** any **hidden value** outside of the **function scope**, such as **another field in the same class** or **global variables**.
    2. It **cannot modify any hidden fields** outside of the **function scope**, such as other **mutable fields in the same class** or **global variables**.
    3. It **cannot depend on any external I/O**. It can't **rely on input** from files, databases, web services, UIs, etc; it can't **produce output**, such as **writing** to a file, database, or web service, **writing** to a screen, etc.

2. A **pure function does not modify** its input parameters.

This is unlike an **OOP** method, which **can depend** on other fields in the same class as the method.

As a result of 1, if a **pure function** is called with an input parameter x an infinite number of times, it will always return the same result y.

A pure function has **no side effects**, meaning that it does not read anything from the **outside world** or write anything to the **outside world**.



by Alvin Alexander
@alvinalexander

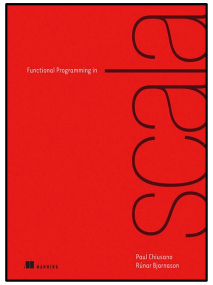
PF = ODI + NSE
Pure **F**unctions **=** **O**utput **D**epends only on **I**nput **+** **N**o **S**ide **E**ffects)

PURITY

FP means programming with **pure functions**, and a **pure function** is one that lacks **side effects**…

A function **f** with input type **A** and output type **B** (written in Scala as a single type: **A => B** , pronounced "**A** to **B** " or "**A** arrow **B** ") is a computation that relates **every value a** of type **A** to **exactly one value b** of type **B** such that **b** is determined solely by the value of **a**. Any changing state of an internal or external process is irrelevant to computing the result **f(a)**. For example, a function intToString having type Int => String will take every integer to a corresponding string. Furthermore, if it really is a function, it will do nothing else.

In other words, a function has no **observable effect** on the execution of the program other than to compute a result given its inputs; we say that it has no **side effects**. We sometimes qualify such functions as **pure functions** to make this more explicit, but this is somewhat redundant.



FP in Scala
by Paul Chiusano
and Runar Bjarnason
@pchiusano @runarorama

Real AI and 14 others Retweeted

john Ⓐ De Goes @jdegoes · 30 Nov 2017

FP is just programming with functions. Functions are:

1. Total: They return an output for every input.
2. Deterministic: They return the same output for the same input.
3. Pure: Their only effect is computing the output.

The rest is just composition you can learn over time.

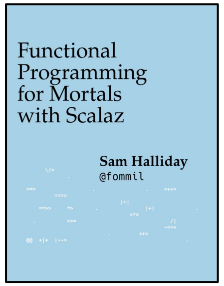
17 254 610

1.2 Pure Functional Programming

Functional Programming is the act of writing programs with **pure functions**. Pure functions have three properties:

- **Total**: return a value for every possible input
- **Deterministic**: return the same value for the same input
- **Inculpable**: no (direct) interaction with the world or program state.

Together, these properties give us an unprecedented ability to reason about our code. For example, input validation is easier to isolate with **totality**, caching is possible when functions are **deterministic**, and interacting with the world is easier to control, and test, when functions are **inculpable**. The kinds of things that break these properties are **side effects**: directly accessing or changing **mutable state** (e.g. maintaining a var in a class or using a legacy API that is **impure**), communicating with **external resources** (e.g. files or network lookup), or throwing **exceptions**.



by Sam Halliday
@fommil

Referential Transparency and Purity

We can formalize this idea of pure functions using the concept of referential transparency (RT). This is a property of expressions in general and not just functions. For the purposes of our discussion, consider an expression to be any part of a program that can be evaluated to a result—anything that you could type into the Scala interpreter and get an answer. For example, 2 + 3 is an expression that applies the pure function + to the values 2 and 3 (which are also expressions). This has no side effect. The evaluation of this expression results in the same value 5 every time. In fact, if we saw 2 + 3 in a program we could simply replace it with the value 5 and it wouldn't change a thing about the meaning of our program.

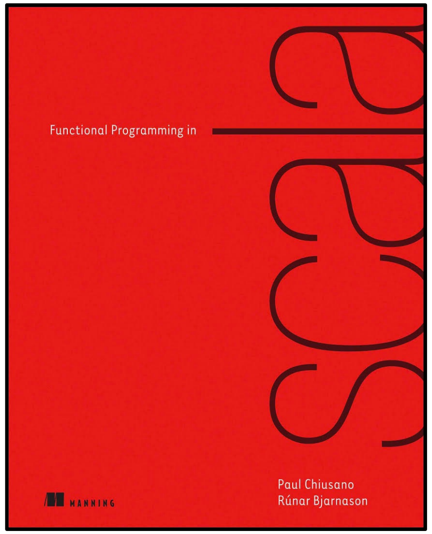
This is all it means for an expression to be referentially transparent—in any program, the expression can be replaced by its result without changing the meaning of the program. And we say that a function is pure if calling it with RT arguments is also RT.

Referential transparency and purity

An expression E is referentially transparent if, for all programs P, all occurrences of E in P can be replaced by the result of evaluating E without affecting the meaning of P.

A function f is pure if the expression f(x) is referentially transparent for all referentially transparent x.[3]

[3] There are some subtleties to this definition, and we'll refine it later in this book. See the chapter notes at our GitHub site (https://github.com/pchiusano/fpinscala; see the preface) for more discussion.



Functional Programming in Scala
(by Paul Chiusano and Runar Bjarnason)
@pchiusano @runarorama

Paradigm

An overall **strategy** or **viewpoint** for doing things. A **paradigm** is a specific **mindset**.

The **object-oriented** **paradigm** is a development strategy based on the concept that systems should be built from a collection of reusable components called objects.

The **structured** **paradigm** is a development strategy based on the concept that a system should be split into two parts: data (modelled using data/persistence model) and functionality (modelled using a process model).

The Object Primer

Functional programming (**FP**) **is a paradigm of programming**, – that is, **an approach that guides programmers to write code in specific ways**, for a wide range of programming tasks.

The **main principle of FP is to write code as a mathematical expression or formula**. **This approach allows programmers to derive code through logical reasoning rather than through guessing**, – similarly to how books on mathematics reason about mathematical formulas and derive results systematically, without **guessing** or "**debugging**." Similarly to mathematicians and scientists who reason about formulas, **functional programmers can reason about code systematically and logically, based on rigorous principles**. This is possible only because **code is written as a mathematical formula**.

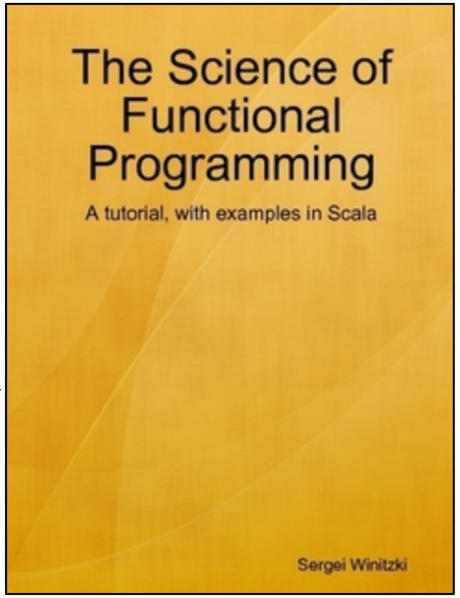
Mathematical intuition is backed by the vast experience accumulated while working with data over thousands of years of human history. **It took centuries to invent flexible and powerful notation such as** $\forall k \in S : p(k)$ **and to develop the corresponding rules of reasoning**. **Functional programmers are fortunate to have at their disposal such a superior reasoning tool**.

As we have seen, **the Scala code for certain computational tasks corresponds quite closely to mathematical formulas**. (**Scala** conventions and syntax, of course, require programmers to spell out certain things that the mathematical notation leaves out.) **Just as in mathematics, large code expressions may be split into parts in a suitable way, so that the parts can be easily reused, flexibly composed together, and written independently from each other**. **The FP community has developed a toolkit of functions (such as .map, .filter, etc.) that proved especially useful in real-life programming**, although many of them are not standard in mathematical literature.

Mastering FP involves practicing to reason about programs as formulas, building up the specific kind of applied mathematical intuition, familiarizing oneself with concepts adapted to programming needs, and learning how to translate the mathematics into code in various cases. The FP community has discovered a number of specific design patterns, founded on mathematical principles but driven by practical necessities of programming rather than by the needs of academic mathematics.
This book explains the required **mathematical principles** in detail, developing them through intuition and practical coding tasks.

Sergei Winitzki

in sergei-winitzki-11a6431



1.7.1 Functional programming as a paradigm

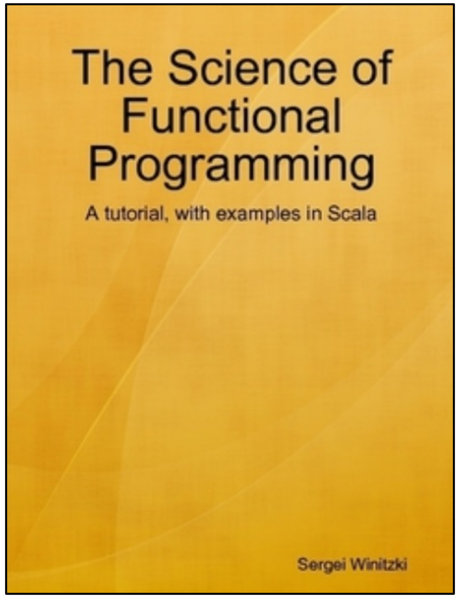
Functional programming (FP) is a paradigm of programming, – that is, an approach that guides programmers to write code in specific ways, for a wide range of programming tasks.

The main principle of FP is to write code as a mathematical expression or formula. This approach allows programmers to derive code through logical reasoning rather than through guessing, – similarly to how books on mathematics reason about mathematical formulas and derive results systematically, without guessing or "debugging." Similarly to mathematicians and scientists who reason about formulas, functional programmers can reason about code systematically and logically, based on rigorous principles. This is possible only because code is written as a mathematical formula.

Mathematical intuition is backed by the vast experience accumulated while working with data over thousands of years of human history. It took centuries to invent flexible and powerful notation such as $\forall k \in S : p(k)$ and to develop the corresponding rules of reasoning. Functional programmers are fortunate to have at their disposal such a superior reasoning tool.

As we have seen, the Scala code for certain computational tasks corresponds quite closely t omathematical formulas. (Scala conventions and syntax, of course, require programmers to spell out certain things that thematehmatical notation leaves out.) Just as inmathematics, large code expressions may be split into parts in a suitable way, so that the parts can be easily reused, flexibly composed together, and written independently from each other. The FP community has developed a toolkit of functions (such as .map, .filter, etc.) that proved especially useful in real-life programming, although many of them are not standard in mathematical literature.

Mastering FP involves practicing to reason about programs as formulas, building up the specific kind of applied mathematical intuition, familiarizing oneself with concepts adapted to programming needs, and learning how to translate the mathematics into code in various cases. The FP community has discovered a number of specific design patterns, founded on mathematical principles but driven by practical necessities of programming rather than by the needs of academic mathematics. This book explains the requiredmathematical principles in detail, developing them through



Sergei Winitzki

sergei-winitzki-11a6431

Definitions of Functional Programming

Functional programming is a way of writing software applications using only **pure functions** and **immutable values**.

A **pure function** is a function that depends only on its declared input parameters and its algorithm to produce its output. It does not read any other values from "the **outside world**" — the world outside of the **function's scope** — and it does not modify any values in the **outside world**

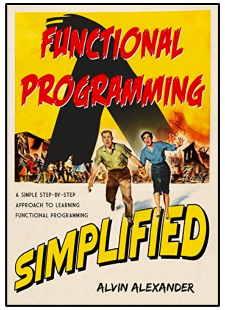
1. A **pure function** depends only on (a) its declared input parameters and (b) its algorithm to produce its result. A **pure function** has no "**back doors**," which means:
    1. Its result **can't depend on reading** any **hidden value** outside of the **function scope**, such as **another field in the same class** or **global variables**.
    2. It **cannot modify any hidden fields** outside of the **function scope**, such as other **mutable fields in the same class** or **global variables**.
    3. It **cannot depend on any external I/O**. It can't **rely on input** from files, databases, web services, UIs, etc; it can't **produce output**, such as **writing** to a file, database, or web service, **writing** to a screen, etc.

2. A **pure function does not modify** its input parameters.

This is unlike an **OOP** method, which **can depend** on other fields in the same class as the method.

As a result of 1, if a **pure function** is called with an input parameter x an infinite number of times, it will always return the same result y.

A pure function has **no side effects**, meaning that it does not read anything from the **outside world** or write anything to the **outside world**.



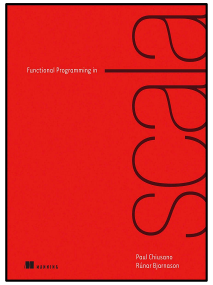
by Alvin Alexander
@alvinalexander

PF = ODI + NSE
Pure **F**unctions = **O**utput **D**epends only on **I**nput + **N**o **S**ide **E**ffects)

FP means programming with **pure functions**, and a **pure function** is one that lacks **side effects**…

A function **f** with input type **A** and output type **B** (written in Scala as a single type: **A => B** , pronounced "**A** to **B** " or "**A** arrow **B** ") is a computation that relates **every value a** of type **A** to **exactly one value b** of type **B** such that **b** is determined solely by the value of **a**. Any changing state of an internal or external process is irrelevant to computing the result **f(a)**. For example, a function intToString having type Int => String will take every integer to a corresponding string. Furthermore, if it really is a function, it will do nothing else.

In other words, a function has no **observable effect** on the execution of the program other than to compute a result given its inputs; we say that it has no **side effects**. We sometimes qualify such functions as **pure functions** to make this more explicit, but this is somewhat redundant.



FP in Scala
by Paul Chiusano
and Runar Bjarnason
@pchiusano @runarorama

Real AI and 14 others Retweeted

john Ⓐ De Goes @jdegoes · 30 Nov 2017

FP is just programming with functions. Functions are:

1. Total: They return an output for every input.
2. Deterministic: They return the same output for the same input.
3. Pure: Their only effect is computing the output.

The rest is just composition you can learn over time.

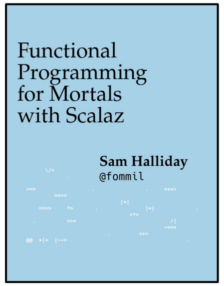
17 254 610

1.2 Pure Functional Programming

Functional Programming is the act of writing programs with **pure functions**. Pure functions have three properties:

- **Total**: return a value for every possible input
- **Deterministic**: return the same value for the same input
- **Inculpable**: no (direct) interaction with the world or program state.

Together, these properties give us an unprecedented ability to reason about our code. For example, input validation is easier to isolate with **totality**, caching is possible when functions are **deterministic**, and interacting with the world is easier to control, and test, when functions are **inculpable**. The kinds of things that break these properties are **side effects**: directly accessing or changing **mutable state** (e.g. maintaining a var in a class or using a legacy API that is **impure**), communicating with **external resources** (e.g. files or network lookup), or throwing **exceptions**.



by Sam Halliday
@fommil

1.2 Exactly what is a (pure) function?

We said earlier that **FP means programming with pure functions**, and **a pure function is one that lacks side effects**. In our discussion of the coffee shop example, we worked off an informal notion of side effects and purity. Here we'll formalize this notion, to pinpoint more precisely what it means to program functionally. This will also give us additional insight into **one of the benefits of functional programming: pure functions are easier to reason about**.

A function `f` with input type `A` and output type `B` (written in Scala as a single type: `A => B`, pronounced "A to B" or "A arrow B") is a computation that **relates every value `a` of type `A` to exactly one value `b` of type `B` such that `b` is determined solely by the value of `a`. Any changing state of an internal or external process is irrelevant to computing the result `f(a)`**. For example, a function `intToString` having type `Int => String` will take every integer to a corresponding string. Furthermore, if it really is a function, it will do nothing else.

In other words, **a function has no observable effect on the execution of the program other than to compute a result given its inputs; we say that it has no side effects. We sometimes qualify such functions as pure functions to make this more explicit, but this is somewhat redundant.** Unless we state otherwise, we'll often use function to imply no side effects.2

We can formalize this idea of pure functions using the concept of referential transparency (RT). This is a property of expressions in general and not just functions. For the purposes of our discussion, consider an expression to be any **part of a program that can be evaluated to a result**—anything that you could type into the Scala interpreter and get an answer. **For example, `2 + 3` is an expression that applies the pure function `+` to the values `2` and `3` (which are also expressions). This has no side effect**. **The evaluation of this expression results in the same value `5` every time. In fact, if we saw `2 + 3` in a program we could simply replace it with the value `5` and it wouldn't change a thing about the meaning of our program**.

This is all it means for an expression to be referentially transparent—in any program, the expression can be replaced by its result without changing the meaning of the program. And we say that **a function is pure if calling it with RT arguments is also RT**. We'll look at some examples next.

[2] Procedure is often used to refer to some parameterized chunk of code that may have **side effects**.
[3] There are some subtleties to this definition, and we'll refine it later in this book. See the chapter notes at our GitHub site (https://github.com/pchiusano/fpinscala; see the preface) for more discussion.

Referential transparency and purity

An expression **`e`** is referentially transparent if, for all programs **`p`**, all occurrences of **`e`** in **`p`** can be replaced by the result of evaluating **`e`** without affecting the meaning of **`p`**.

A function `f` is pure if the expression `f(x)` is referentially transparent for all referentially transparent `x`.[3]

A function `f` with input type `A` and output type `B` is a computation that **relates every value `a` of type `A` to exactly one value `b` of type `B` such that `b` is determined solely by the value of `a`. Any changing state of an internal or external process is irrelevant to computing the result `f(a)`**

In other words, **a function has no observable effect on the execution of the program other than to compute a result given its inputs; we say that it has no side effects. We sometimes qualify such functions as pure functions to make this more explicit, but this is somewhat redundant.**

The evaluation of this expression results in the same value `5` every time.

We can formalize this idea of pure functions using the concept of referential transparency (RT).

This is all it means for an expression to be referentially transparent—in any program, the expression can be replaced by its result without changing the meaning of the program. And we say that **a function is pure if calling it with RT arguments is also RT**.

FP means programming with pure functions, and **a pure function is one that lacks side effects**

one of the benefits of functional programming: pure functions are easier to reason about.

A function `f` with input type `A` and output type `B` is a computation that relates every value `a` of type `A` to exactly one value `b` of type `B` such that `b` is determined solely by the value of `a`. Any changing state of an internal or external process is irrelevant to computing the result `f(a)`.

Pinned Tweet

John Ⓐ De Goes @jdegoes · 30 Nov 2017

FP is just programming with functions. Functions are:

1. Total: They return an output for every input.
2. Deterministic: They return the same output for the same input.
3. Pure: Their only effect is computing the output.

The rest is just composition you can learn over time.

15 204 511

GitHub, Inc. [US] | https://github.com/fommil/fpmortals/blob/master/manuscript/introduction.md

Pure Functional Programming

Functional Programming is the act of writing programs with *pure functions*. Pure functions have three properties:

- **Total:** return a value for every possible input
- **Deterministic:** return the same value for the same input
- **Inculpable:** no (direct) interaction with the world or program state.

A function `f` with input type `A` and output type `B` is a computation that relates every value `a` of type `A` to exactly one value `b` of type `B` such that `b` is determined solely by the value of `a`. Any changing state of an internal or external process is irrelevant to computing the result `f(a)`.

a function has no observable effect on the execution of the program other than to compute a result given its inputs; we say that it has no side effects. We sometimes qualify such functions as pure functions to make this more explicit, but this is somewhat redundant.

FP means programming with pure functions, and a pure function is one that lacks side effects

Pinned Tweet

John Ⓐ De Goes @jdegoes · 30 Nov 2017

FP is just programming with functions. Functions are:

1. Total: They return an output for every input.
2. Deterministic: They return the same output for the same input.
3. Pure: Their only effect is computing the output.

The rest is just composition you can learn over time.

15 204 511

GitHub, Inc. [US] | https://github.com/fommil/fpmortals/blob/master/manuscript/introduction.md

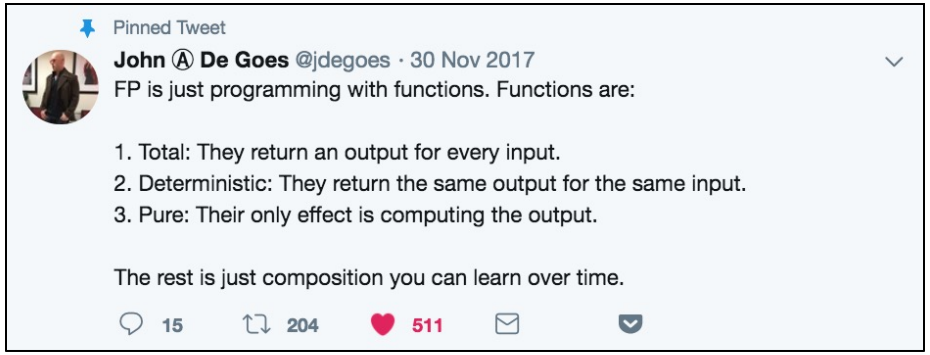
Pure Functional Programming

Functional Programming is the act of writing programs with *pure functions*. Pure functions have three properties:

- **Total:** return a value for every possible input
- **Deterministic:** return the same value for the same input
- **Inculpable:** no (direct) interaction with the world or program state.

A function `f` with input type `A` and output type `B` is a computation that relates every value `a` of type `A` to exactly one value `b` of type `B` such that `b` is determined solely by the value of `a`. Any changing state of an internal or external process is irrelevant to computing the result `f(a)`.
…In other words, a function has no observable effect on the execution of the program other than to compute a result given its inputs; we say that it has no side effects. We sometimes qualify such functions as pure functions to make this more explicit, but this is somewhat redundant.

FP means programming with pure functions, and a pure function is one that lacks side effects



Pinned Tweet

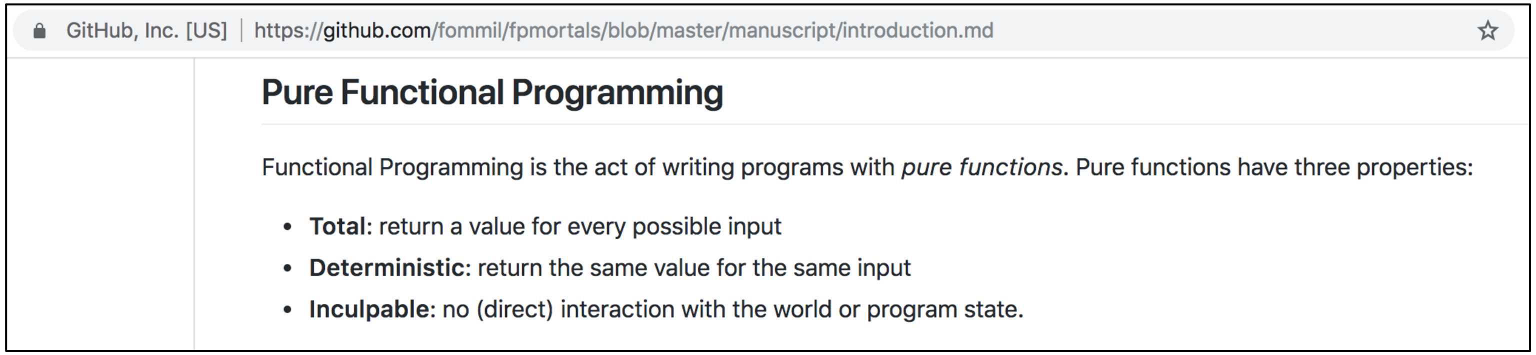
John Ⓐ De Goes @jdegoes · 30 Nov 2017

FP is just programming with functions. Functions are:

1. Total: They return an output for every input.
2. Deterministic: They return the same output for the same input.
3. Pure: Their only effect is computing the output.

The rest is just composition you can learn over time.

15 204 511



GitHub, Inc. [US] | https://github.com/fommil/fpmortals/blob/master/manuscript/introduction.md

Pure Functional Programming

Functional Programming is the act of writing programs with *pure functions*. Pure functions have three properties:

- **Total:** return a value for every possible input
- **Deterministic:** return the same value for the same input
- **Inculpable:** no (direct) interaction with the world or program state.

Real AI and 14 others Retweeted

john Ⓐ De Goes @jdegoes 30 Nov 2017

FP is **just programming** with functions. Functions are:

1. Total: They return an output for every input.
2. Deterministic: They return the same output for the same input.
3. Pure: Their only effect is computing the output.

T… **t composition** you can learn over time.

john Ⓐ De Goes

FP is **just programming** with functions. Functions are:

1. Total: They return an output for every input.
2. Deterministic: They return the same output for the same input.
3. Pure: Their only effect is computing the output.

The rest is **just composition** you can learn over time.

17 replies .254 retweets610 likes

1.2 Pure Functional Programming

Functional Programming is the act of writing programs with pure functions. Pure functions have three properties:

- **Total**: return a value for every possible input
- **Deterministic**: return the same value for the same input
- **Inculpable**: no (direct) interaction with the world or program state.

Together, these properties give us an unprecedented ability to reason about our code. For example, input validation is easier to isolate with totality, caching is possible when functions are deterministic, and interacting with the world is easier to control, and test, when functions are inculpable.

The kinds of things that break these properties are side effects: directly accessing or changing mutable state (e.g. maintaining a var in a class or using a legacy API that is impure), communicating with external resources (e.g. files or network lookup), or throwing exceptions.

GitHub, Inc. [US] | https://github.com/fommil/fpmortals/blob/master/manuscript/introduction.md

Pure Functional Programming

Functional Programming is the act of writing programs with *pure functions*. Pure functions have three properties:

- **Total**: return a value for every possible input
- **Deterministic**: return the same value for the same input
- **Inculpable**: no (direct) interaction with the world or program state.

inculpable adjective

in·cul·pa·ble | \(ˌ)in-ˈkəl-pə-bəl \

Definition of *inculpable*

: free from guilt : BLAMELESS

Pinned Tweet

John Ⓐ De Goes @jdegoes · 30 Nov 2017

FP is just programming with functions. Functions are:

1. Total: They return an output for every input.
2. Deterministic: They return the same output for the same input.
3. Pure: Their only effect is computing the output.

The rest is just composition you can learn over time.

15 204 511

I think you can define FP with just two statements:
1. FP is about writing software applications using only pure functions.
2. When writing FP code you only use immutable values — val fields in Scala.
And when I say "only" in those sentences, I mean only.

You can combine those two statements into this simple definition:

> Functional programming is a way of writing software applications using only pure functions and immutable values.

Of course that definition includes the term "pure functions," which I haven't defined yet, so let me fix that.

A working definition of "pure function"

I provide a complete description of pure functions in the "Pure Functions" lesson, but for now, I just want to provide a simple working definition of the term.

A pure function can be defined like this:
- The output of a pure function depends only on (a) its input parameters and (b) its internal algorithm.
  - This is unlike an OOP method, which can depend on other fields in the same class as the method.
- A pure function has **no side effects**, meaning that it does not read anything from the outside world or write anything to the outside world.
  - It does not read from a file, web service, UI, or database, and does not write anything either.
- As a result of those first two statements, if a pure function is called with an input parameter x an infinite number of times, it will always return the same result y.
  - For instance, any time a "string length" function is called with the string "Alvin", the result will always be 5.

As I mentioned in the “What is Functional Programming?” chapter, I define functional programming (FP) like this:

> Functional programming is a way of writing software applications using only pure functions and immutable values.

Because that definition uses the term “pure functions,” it’s important to understand what a pure function is. I gave a partial pure function definition in that chapter, and now I’ll provide a more complete definition.

1. A pure function depends only on (a) its declared input parameters and (b) its algorithm to produce its result. A pure function has no “back doors,” which means:
    1. Its result can’t depend on reading any hidden value outside of the function scope, such as another field in the same class or global variables.
    2. It cannot modify any hidden fields outside of the function scope, such as other mutable fields in the same class or global variables.
    3. It cannot depend on any external I/O. It can’t rely on input from files, databases, web services, UIs, etc; it can’t produce output, such as writing to a file, database, or web service, writing to a screen, etc.

2. A pure function does not modify its input parameters.

This can be summed up concisely with this definition:

> A pure function is a function that depends only on its declared input parameters and its algorithm to produce its output. It does not read any other values from “the outside world” — the world outside of the function’s scope — and it does not modify any values in the outside world.