Distinctive traits of the functional languages

Asked 15 years, 11 months ago Modified 12 years, 8 months ago

Viewed 3k times

19

It is known that all functional languages share some basic properties like using functions as basic building block for programs with all the consequences like using recursion instead of iteration. However, some fundamental differences also exist. Lisp uses a single representation for both Lisp code and data, while ML has no standard representation of ML code. Erlang has a built-in actor-based concurrency. Haskell has monads. Haskell makes a distinction in the static type system between pure and impure functions; ML does not.

What are the distinctive **fundamental** differences between other functional languages (Clojure, F#, Arc, any other)? By fundamental I mean something which influences the way you develop in this language, and *not* for example, whether it is integrated with some wide-spread runtime.

haskell f# functional-programming erlang clojure

edited Jan 30, 2009 at 3:02

Follow

Norman Ramsey
202k ● 62 ● 371 ● 541

asked Jan 27, 2009 at 21:42

Sergey Mikhanov
8,950 ● 10 ● 45 ● 54

Most functional languages *do* have a distinction between code and data. Lisp is atypical in this respect. – Iraimbilanja Jan 27, 2009 at 21:44

My bad, haven't looked deep enough and made this wrong conclusion of Lisp and Erlang. – Sergey Mikhanov Jan 27, 2009 at 21:50

I am unclear what the question is. Are you asking e.g. "what sets F# apart from Clojure", or else what are you asking? Are you looking for information about specific languages, or FP languages in general? – Brian Jan 28, 2009 at 2:38

Yep, exactly something like "what sets F# apart from Clojure". E.g. Haskell being set apart from Lisp by having monads. – Sergey Mikhanov Jan 28, 2009 at 11:54

Monads are distinctive, but I think of them as a side-effect of more fundamental traits: monads become useful in a lazy, pure language, and require higher-order datatypes. – Chris Conway Jan 28, 2009 at 14:23

7 Answers

Sorted by: Highest score (default)

25

Off the top of my head:

- *lazy vs. eager* (aka *non-strict* vs. *strict* or *call-by-need* vs. *call-by-value*): are function arguments




evaluated before the function application, or after, or never?

- *pure vs. impure*: does the language allow functions to have side effects? Does it have mutable references?
- *static vs. dynamic*: does the language check types at compile time or runtime?
- *algebraic datatypes*: does the language support pattern matching over variant types?
- *metaprogramming*: does the language provide a powerful code generation system?
- *concurrency* and *parallelism*: are threads/processes a first-class abstraction? Does the language make it easy to run multiple computations at the same time?
- *"exotic" types*: how expressive is the static type system? GADTs? Dependent types? Linear types? System F?

Only the first two items are really unique to functional languages (i.e., almost all imperative languages are eager and impure).

Share Improve this answer

Follow

edited Dec 9, 2011 at 1:04

Lambda Fairy

14.6k ● 7 ● 43 ● 71

answered Jan 28, 2009 at 1:54

Chris Conway

56k ● 43 ● 131 ● 155

Don't throw "strong/weak" in with "static/dynamic" typing, make it a point of itself. – Svante Jan 30, 2009 at 3:18

1 Strong vs. weak is ill-defined. I consider static strong and dynamic weak. YMMV. – Chris Conway Jan 30, 2009 at 4:29

23 Static typing just means that types are determined at compile time, dynamic typing means that types are determined at run time. Strong typing means that type mismatches are always errors, weak typing allows implicit conversion. These distinctions are orthogonal. For example, C has weak static typing, Common Lisp has strong dynamic typing, Perl has weak dynamic typing, and Haskell has strong static typing. – Svante May 6, 2009 at 3:30

@Svante I've fixed the terminology in the answer -- it'll come up once someone approves it. – Lambda Fairy Dec 9, 2011 at 1:03

14

I like Chris Conway's answer that states some important axes that help classify different functional languages.

In terms of features of specific languages, I'll pick **F#** to call out some features not found in many other FPLs:

- **Active Patterns**: a number of FPLs have algebraic data types and pattern-matching, but the F# feature called 'active patterns' lets you define new patterns that allow you to use pattern-matching syntax on arbitrary data.
- **Computation expressions**: F# has some beautiful syntactic sugar for authoring monadic code; though the type system cannot express higher-kinded

polymorphism (no abstraction over type constructors) so you can't write code for an arbitrary monad M, the code you can write for a fixed monad is very cool, and people write some great comprehensions in the seq{} or async{} monads.

- **Quotations**: the usual 'code as data for metaprogramming' bit, though F# has an expressive static type system and rich syntax, and I'm not sure how many non-lisps can do this.

In terms of general classification, F# is

- **eager** (strict, call-by-value; but 'lazy' is a keyword & library and using seq/IEnumerable for some laziness is a common strategy)
- **impure** (though syntax biases you towards a purer-by-default style)
- **static** (with type inference, so F# often 'feels like scripting', only with type safety)

Your question is phrased in a way with clear bias against some extra-language pragmatics (e.g. what runtime does it integrate with), but you also ask what "influences the way you develop", and these things do influence that:

- Visual Studio integration means a **great editing experience** (e.g. Intellisense)
- Visual Studio integration means a **great debugging experience** (e.g. breakpoints/tracepoints, locals, immediate window, ...)

- **REPL** for scripting or UI-on-the-fly is hotness (fsi.exe command-line, or "F# Interactive" integrated in VS)
- .NET integration means for most 'X' there's already a library to do that
- side tools like FsLex/FsYacc, and integration with MSBuild which makes 'build system' easy

(I think that trying to separate a language from its runtime and tooling is a mostly academic exercise.)

So there's a description of lot of distinctive features of one particular language of which I am a fan. I hope others might post similar answers that call out distinctive features of other individual languages.

edited Jun 20, 2020 at 9:12

Community Bot

1 ● 1

answered Jan 28, 2009 at 13:43

Brian

119k ● 17 ● 243 ● 304

Objective Caml and Haskell both have very interesting quasiquoting mechanisms. – Norman Ramsey Jan 30, 2009 at 3:03

Scala has excellent support for custom pattern matching. Scala, Clojure and Haskell have the build systems sbt, Leiningen and Cabal respectively. Metaprogramming is supported with MetaOCaml and Template Haskell. Other than the Visual Studio integration, F#'s features aren't at all unique. – Lambda Fairy Dec 9, 2011 at 1:12

1. Non-strict vs strict evaluation.
2. Static vs dynamic typing.
3. Structural vs nominal static typing. OCaml is the only language I can think of with structural typing (in both objects and polymorphic variants), which closes the gap with dynamic typing by removing the need to define many types (e.g. variant types).
4. Hindley-Milner derivatives vs other static type inference algorithms. SML, OCaml, Haskell and F# use type inference algorithms based upon Hindley-Milner whereas Scala has only local type inference (like C# 3) and requires many more annotations to compile. (Haskell code is often full of type annotations at the function level but most are unnecessary and are added for documentation and to help the compiler in the presence of errors).
5. Pattern matching vs manual deconstruction. SML, OCaml, F#, Haskell, Mathematica and Scheme

automate the deconstruction of values.

6. Closed sum types vs only open sum types. SML, OCaml, F# and Haskell allow closed/sealed algebraic types to be defined to strengthen static typing by conveying more specific constraints implicitly. OCaml and F# also allow open sum types whereas SML does not and Haskell requires an elaborate workaround (described by Oleg Kiselyov).
7. Bounded-time patterns. Pattern matching is very fast in SML and (vanilla) OCaml but has unknown performance in F# due to active patterns and even unknown asymptotic complexity in Mathematica.
8. On-the-fly compilation to native code. F#, Lisp and Scheme allow code to be generated, compiled and executed efficiently at run-time.
9. Macros. OCaml, Mathematica, Lisp and Scheme are extensible languages.
10. Standardized vs proprietary. SML, Haskell 2010, Common Lisp and Scheme are standardized languages whereas OCaml, Erlang, F# and Mathematica are proprietary.

edited Mar 31, 2012 at 14:23

alternative
12.9k ● 5 ● 43 ● 41

answered May 6, 2009 at 3:05

J D
48.6k ● 14 ● 174 ● 277

1 Huh? OCaml and Erlang are proprietary? – Longpoke Dec 16, 2010 at 5:10

3 I think Jon means "does not have a published standard", not proprietary in their implementations. – Lambda Fairy Dec 9, 2011 at 1:14

1 "proprietary" does not equal "not standardized". These are orthogonal attributes. Plenty of open-source languages are non-proprietary (by definition) but do not have published standards. You can also have proprietary languages that have published standards (C# on .Net being a prominent example) – mikera Sep 30, 2012 at 3:24

5

There are many differences but **only two differences I'd categorize as fundamental** in that they make a big difference to your development:

1. Dynamically typed vs static, polymorphic type system with algebraic data types and type inference. A static type system restricts code somewhat, but has many advantages:
    - Types are documentation that is checked by the compiler.
    - The type system helps you choose what code to write next, and when you're not sure just what to write, the type system helps you easily and quickly rule out many alternatives.
    - A powerful, modern, polymorphic type system is unreasonably good at detecting small, silly, time-wasting bugs.

2. Lazy evaluation as the default everywhere vs lazy evaluation restricted to carefully controlled constructs.
    - Lazy vs eager has tremendous implications for your ability to predict and understand the time and space costs of your programs.
    - In a fully lazy language, you can completely decouple *production* of data from decisions about what to do with data once produced. This is especially important for search problems as it becomes much easier to modularize and reuse code.

Share Improve this answer

Follow

answered Jan 30, 2009 at 3:10

Norman Ramsey
202k ● 62 ● 371 ● 541

3

Functional Programming is a style, not a language construct

Most functional languages have some common principles:

- Immutable objects
- Closures and anonymous functions
- Generic algorithms
- Continuations

But the most important principle is that they usually force you to write in a functional style. You can program in a functional style in most any language. C# could be considered "functional" if you write code like that, as could any other language.

Share Improve this answer

Follow

edited Feb 27, 2009 at 19:53

GEOCHET

21.3k ● 15 ● 77 ● 99

answered Jan 27, 2009 at 21:47

Alex Fort

18.8k ● 5 ● 44 ● 51

modules and functors are not necessary, but common as well. – nlucaroni Jan 27, 2009 at 21:54

I would include functions as a value. – rampion Jan 27, 2009 at 21:56

Yes, but I wonder what are distinctive properties between different functional languages, and not between functional and non-functional ones. – Sergey Mikhanov Jan 27, 2009 at 22:00

well, definitly, functions as value as rampion mentions. Type systems, pattern matching, and list comprehension are other good character as well. – nlucaroni Jan 27, 2009 at 22:11

Depends who you ask. I would say that first-class lexical closures make a language functional. Haskell programmers often use the completely different idea that functional programming means only purely functional, i.e. essentially just Haskell today. – J D May 6, 2009 at 2:34

2

Fundamental properties?

- Functional Purity (lack of side-effects)
- As a tie-in from the above, lack of state.
- Pattern-matching in functions

The first is beautiful, the second is an ugly side-effect of the former (pun intended).

The real-world compensation for lack-of-state is what I find to be the biggest differentiator between functional languages.

Those few things give lots of freebies. Most of the time, languages handle memoization.

Share Improve this answer

Follow

answered Jan 27, 2009 at 21:47

user54650

4,406 ● 2 ● 25 ● 27

but most FP languages aren't pure – Mauricio Scheffer Jan 27, 2009 at 23:37

The extent to which a language can be said to facilitate FP is directly proportional to its purity. – Apocalisp Jan 27, 2009 at 23:52

there isn't a lack of state, it's just that state is stored elsewhere, often on the stack. – apg Jan 28, 2009 at 0:56

1 Hmm, the extent a language is "functional", by pure definition, is mainly related to it's higher order function/functions as value capabilities. If a language makes

it hard to define and return functions, then it's less functional, no? – MichaelGG Jan 28, 2009 at 4:49

2

When you say code as data you are referring to a language where the code is represented in a data structure. This is referred to as Homoiconicity and it usually only true for languages that are lisp dialects or something close to it. Haskell, Erlang and Scala are not Homoiconic, Clojure is.

Clojure's fundamental differentiators are:

1. It has a Software Transactional Memory system, which makes shared state concurrent programming easier
2. It is a Lisp, unlike Haskell or Erlang, therefore all code is data, which allows you to make what look likes changes to the language itself at runtime through the macro system
3. It runs on the JVM, which means you have direct access to all Java libraries
4. Clojure data structures implement Java interfaces such as Collection, List, Map, Runnable and Callable where appropriate. Strings are just Java Strings, Numbers are Java Integers and Doubles. This means Clojure data structures can be passed directly to Java libraries without any bridging or translation

Share Improve this answer answered Jan 28, 2009 at 1:12

Follow

pjb3

5,283 ● 5 ● 29 ● 44

2 Code is data != Homoiconicity. For example, camlp4 provides code as data but it is not necessarily homoiconic because it support arbitrary syntaxes. – J D May 6, 2009 at 2:38

Haskell does have modules for STM: en.wikipedia.org/wiki/Software_transactional_memory#Haskell – mb21 Jul 8, 2012 at 18:45