An Introduction to Effectful Programming in Flix

Magnus Madsen

Today

You may be familiar with **imperative programming**.

You may be familiar with **object-oriented programming**.

You may be familiar with **functional programming**.

Today: effect-oriented programming in Flix

If you love types, you are going to love effects!

Flix Team

Magnus

Matthew

Jonathan

Andreas

Jakob

Open Source Contributors

Adam Yasser Tallouzi
Alexander Dybdahl Troelsen
Andreas Heglingegård
Anna Blume Jakobsen
Anna Krogh
Beinir Ragnuson
Benjamin Dahse
Casper Dalgaard Nielsen
Chanattan Sok
Chenhao Gao
Christian Bonde
Daniel Anker Hermansen
Daniel Welch
Darius Tan
Dylan Do Amaral
Erik Funder Carstensen
Erik Kruuse
Esben Bjerre
Felix Berg
Felix Wiemuth
Frederik Arp Frandsen
Frederik Kirk Kristensen
Herluf Baggesen
Holger Dal Mogensen
Ifaz Kabir
J. Ryan Stinnett
Jacob Harris Cryer Kragh
Jason Mittertreiner
Jesper Skovby
Jim Zhang
Joseph Tan
Justin Fargnoli
Kengo TODA
Liam Palmer
Lionel Mendes
Lukas Rønn
Luqman Aden
Magnus Holm Rasmussen
Maksim Gusev
Manoj Kumar
Marcus Bach
Miguel Angelo Nicolau Fialho
Ming-Ho Yee
Nada Amin
Nathan Bedell
Nicola Dardanis
Nina Andrup Pedersen
Ondřej Lhoták
Oskar Haarklou Veileborg
Patrick Bering Tietze
Patrick Lundvig
Paul Butcher
Paul Phillips
Quentin Stiévenart
Ramin Zarifi
Ramiro Calle
Rasmus Larsen
Roland Csaszar
Sam Ezeh
Simon Dalgas Christensen
Simon Meldahl Schmidt
Stephen Bastians
Stephen Tetley
Surya Somayyajula
Thomas Søe Plougsgaard
Xavier deSouza
Yisrael Union
Yukang Xie
Ziyao Wei

1 Effect-Oriented Programming

What is an Effect System?

An **effect system** aims to describe the ***actions*** of a program.

- Does this function read from the file system?
- Does this function access the network?
- Does this function mutate memory in the heap?

We can use effect systems

1. to **support program reasoning**
2. to **enforce safety properties**
3. to **enable compiler optimizations**

Type and Effect Systems, Pictorially

Here is a simple function:

```
def f(x) = x / getCurrentMinute()
```

What can be said about this function?

- A **type system** tells us that `x` has type `Int` and `f` has type `Int -> Int`
- An **effect system** tell us that `f` may have the effects `{DivByZero, NonDet}`.

Purity (1/2)

We can express that a function is pure:

```
def add(x: Int32, y: Int32): Int32 \ { } = ...
                                    // ^^^ empty set effect
```

Here the implementation of `add` cannot have any side-effects.

Purity (2/2)

We can also require that a function argument is pure:

```
def count(f: a -> Bool \ { }, l: List[a]): Int32 \ { } = ...
                             // ^^^ empty effect set
```

Here `f` cannot have any effects.

Effectful Functions

We can also write a function with a specific effect:

```
def sayHello(name: String): Unit \ { IO } =
    println("Hello ${name}!")     // ^^ printing is impure
```

The `IO` effect describes an action that interacts with the outside world.

Effect Safety

We **cannot** subvert the type and effect system.

For example, if we write:

```
def helloWorld(): Unit \ { } =
    println("Hello World!")
```

The Flix compiler reports:

```
>> Unable to unify the effects: 'Pure' and 'IO'.

2 |     println("Hello World!")
        ^^^^^^^^^^^^^^^^^^^^^^^
        mismatched effects.
```

Effect Polymorphism

We can express that the effects of function depend on its argument:

```
def map(f: a -> b \ ef, l: List[a]): List[b] \ ef = ...
                     // ^^ effect variable         ^^ effect variable
```

The effects of `map` are the same as the effects of `f`:

```
List.map(x -> x * x + 42, l) // has the effect { }
List.map(x -> println(x), l) // has the effect { IO }
```

Function Composition

We can compose two functions:

```
def >>(f: a -> b \ ef1, g: b -> c \ ef2): a -> c \ ef1 + ef2 = ...
                                       // effect union ^^^^^^^^^
```

The composed function has the effects of `f` and `g`.

For example:

- If `f` has effect `{}` and `g` has effect `{IO}` then the result is `{IO}`.
- If `f` has effect `{NonDet}` and `g` has effect `{IO}` then the result is `{NonDet, IO}`.

Effect Exclusion

We can express a function that excludes a specific effect:

```
def onException(f: Exception -> Unit \ ef - {Throw}): Unit = ...
```

Here `onException` can be called with any function that does not throw.

As another example:

```
def onMouseDown(f: MouseEvent -> Unit \ ef - {Block}): Unit = ...
```

Four Kinds of Effects

Flix has four categories of effects:

- Primitive
- Heap
- Library-Defined
- User-Defined

Primitive Effects

In Flix, the current primitive effects are:

Env	**Exec**	**FsRead**	**FsWrite**
Net	**NonDet**	**Sys**	**IO**

2 Heap Effects

Local Mutable Memory

Key Idea: If a function uses mutable memory “local to that function” then we can view it as being pure.

Example: Sorting

```
///
/// Sort the given list `l` so that elements
/// are ordered from low to high according
/// to their `Order` instance.
///
def sort(l: List[a]): List[a] with Order[a] =
    region rc {
        let arr = List.toArray(rc, l);
        Array.sort(arr);
        Array.toList(arr)
    }
```

1. Introduce a new region.
2. Allocate (mutable) data in the region.
3. Do imperative programming.
4. Return immutable data.

Upshot: Using an array-based sort is much faster than any list-based sort.

Example: MutList

Here is how we can use a `MutList[t, r]`:

```
def main(): Unit \ IO =
    region rc {
        let animals = MutList.empty(rc);    // Heap[rc]
        MutList.push("Elephant", animals);  // Heap[rc]
        MutList.push("Giraffe", animals);   // Heap[rc]
        MutList.push("Zebra", animals);     // Heap[rc]
        println(MutList.pop(animals))       // Heap[rc] + IO
    } // IO
```

Prints `Some("Zebra")`.

Example: MutList (2/2)

The API of `MutList` is:

```
mod MutList {
    def empty(rc: Region[r]): MutList[a, r] \ Heap[r]

    def push(x: a, v: MutList[a, r]): Unit \ Heap[r]

    def pop(v: MutList[a, r]): Option[a] \ Heap[r]

    def count(f: a -> Bool \ ef, v: MutList[a, r]): Int32 \ ef + Heap[r]
}
```

Summary: Local Mutable Memory

We can use *mutable memory* inside pure functions. Allows us to:

- implement functions in imperative style.
- use an imperative style when it is more natural and/or more efficient.

We can be functional programmers but use imperative style when we want!

We get the best of both worlds!

3 Associated Effects

Adding Numbers

We can write a function to add two integers:

```
def add(x: Int32, y: Int32): Int32 = ...
```

We can also write a function to add two floating-points:

```
def add(x: Float32, y: Float32): Float32 = ...
```

Abstracting over Addition

We can abstract over addition with a trait (type class):

```
trait Add[t] {
    def add(x: t, y: t): t
}

instance Add[Int32] {
    def add(x: Int32, y: Int32): Int32 = ...
}

instance Add[Float32] {
    def add(x: Float32, y: Float32): Float32 = ...
}
```

Upshot: We can reuse the `+` symbol as an alias for `Add.add`.

The Problem

What happens when we get to **division?**

```
def div(x: Int32, y: Int32): Int32 \ {DivByZero} = ...
                                //   ^^^^^^^^^^^ potential exception
```

But also:

```
def div(x: Float32, y: Float32): Float32 = ...
```

Oops!

The **effect behavior** of integer and floating-point division is **different!**

Q: How can we write a common abstraction for division?

Associated Effects

We can use an **associated effect**!

- An associated effect is an *abstract effect member* of a trait.

We change the `Div` trait to:

```
trait Add[t] {
    type Aef: Eff // associated effect member
    def add(x: t, y: t): t \ Aef
}
```

Back to Division

We can now write implementations for integer and floating-point division:

```
instance Add[Int32] {
    type Aef = {DivByZero}
    def add(x: Int32, y: Int32): Int32 \ DivByZero = ...
}
```

```
instance Add[Float32] {
    type Aef = {}
    def add(x: Float32, y: Float32): Float32 \ {} = ...
}
```

Indexable

```
trait Indexable[t] {
    type Idx: Type
    type Elm: Type
    type Aef: Eff
    def get(t: t, i: Idx): Elm \ Aef
}
```

```
instance Indexable[List[t]] {
    type Idx = Int32
    type Elm = t
    type Aef = {OutOfBounds}
}
```

```
instance Indexable[MutMap[k, v, r]] {
    type Idx = k
    type Elm = v
    type Aef = {Heap[r], OutOfBounds}
}
```

Summary: Associated Effects

An **associated effect** is an **abstract effect member** of a trait.

Each trait instance specifies the effect.

Associated effects arise when abstracting over:

- partial and total functions
- immutable and mutable data
- resources

4 Effects and Handlers

Programming with Effect Handlers

Write **one** abstract program.

- Express indirect inputs (e.g. current time) as an effect
- Express indirect outputs (e.g. writing to a file) as an effect

The type-and-effect system tracks these effects.

Install different handlers

- One for production
- One for testing
- More for adapting to different APIs

Enjoy your **modular**, **reusable**, **testable** implementation!

Example: A Small Http Client (1/2)

```
def main(): Unit \ {Net, IO} =
    run {
        let url = "http://example.com/";
        Logger.info("Downloading URL: '${url}'");
        match HttpWithResult.get(url, Map.empty()) {
            case Result.Ok(response) =>
                let file = "data.txt";
                Logger.info("Saving response to file: '${file}'");
                let body = Http.Response.body(response);
                match FileWriteWithResult.write(str = body, file) {
                    case Result.Ok(_) =>
                        Logger.info("Response saved to file: '${file}'")
                    case Result.Err(err) =>
                        Logger.fatal("Unable to write file: '${err}'")
                }
            case Result.Err(err) =>
                Logger.fatal("Unable to download URL: '${err}'")
        }
    } with FileWriteWithResult.runWithIO
      with HttpWithResult.runWithIO
      with Logger.runWithIO
```

Example: A Small Http Client (2/2)

```
def main(): Unit \ {Net, IO} =
    run {
        // ...
        // ... as before ...
        // ...
    } with FileWriteWithResult.runWithIO
      with Logger.runWithIO
      with handler HttpWithResult {
          def request(_method, _url, _headers, _body, resume) = {
              let e = IoError(ErrorKind.ConnectionFailed, "Oops!");
              resume(Err(e))
          }
      }
```

Effect handlers work like **resumable exceptions**.

Advantages of Effect Handlers

Effects and handlers can be used to support **modularity**, **reusability**, and **testability**.

Effects in Flix

Flix: An **effect-oriented** programming language.

Recap:

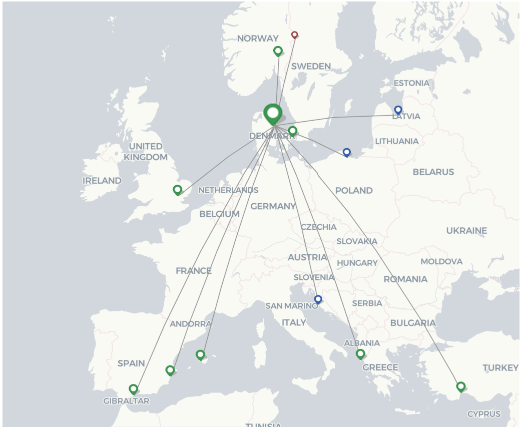
- primitive effects, heap effects, and effects with handlers
- effect polymorphism and effect exclusion
- associated effects

5 But wait, there is more...

Graph Queries

Motivation: I want to go on vacation, but where can I go?

I can fly from Aarhus airport to a few airports in Europe. From there I can continue my journey.



Example: Embedded Datalog

We want to solve a classic graph reachability problem.

We can do so elegantly using Flix's support for **embedded Datalog:**

```
///
/// Computes all airports reachable from origin.
///
def reachable(origin: String, routes: List[(String, String)]): List[String] =
    let db = inject routes into Route;
    let pr = #{
        Path(src, dst) :- Route(src, dst).
        Path(src, dst) :- Path(src, hop), Route(hop, dst).
    };
    query db, pr select dst from Path(origin, dst) |> Foldable.toList
```

We can easily extend this program with more constraints.

Summary: Embedded Datalog

Flix supports embedded Datalog programs as first-class values.

- We can implement functions using `inject` and `query`.
- Datalog with negation is a very expressive logic language.
- Embedded Datalog programs are fully integrated into the language.

Upshot: We can use Datalog where it really shines: to answer graph queries.

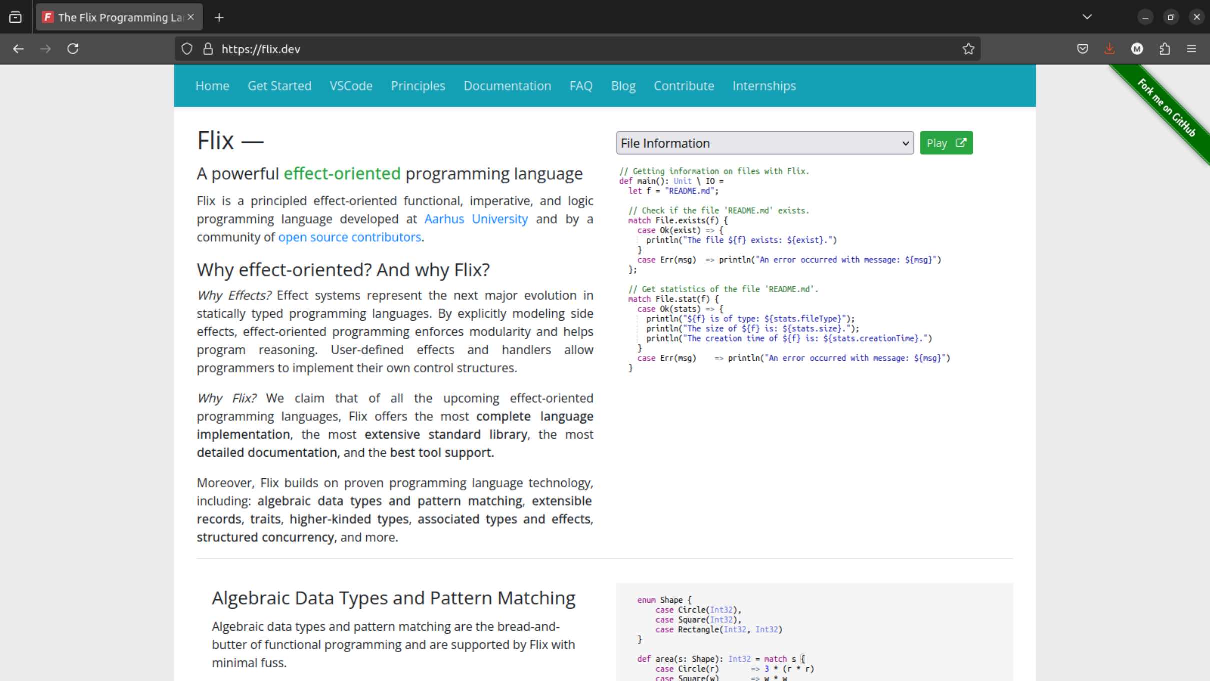
Reflections on Programming in Flix

Flix allows functions to be written in the most natural and/or efficient style:

- **Functionally** (i.e. with immutable data structures)
- **Imperatively** (i.e. with mutable data structures)
- **Declaratively** (i.e. as a collection of logic constraints)

... without revealing these implementation to the clients.

6 Ecosystem and Tooling


The Flix Programming La
https://flix.dev
Home Get Started VSCode Principles Documentation FAQ Blog Contribute Internships
Fork me on GitHub
File Information
Play

Flix —

A powerful effect-oriented programming language

Flix is a principled effect-oriented functional, imperative, and logic programming language developed at Aarhus University and by a community of open source contributors.

Why effect-oriented? And why Flix?

Why Effects? Effect systems represent the next major evolution in statically typed programming languages. By explicitly modeling side effects, effect-oriented programming enforces modularity and helps program reasoning. User-defined effects and handlers allow programmers to implement their own control structures.

Why Flix? We claim that of all the upcoming effect-oriented programming languages, Flix offers the most **complete language implementation**, the most **extensive standard library**, the most **detailed documentation**, and the **best tool support.**

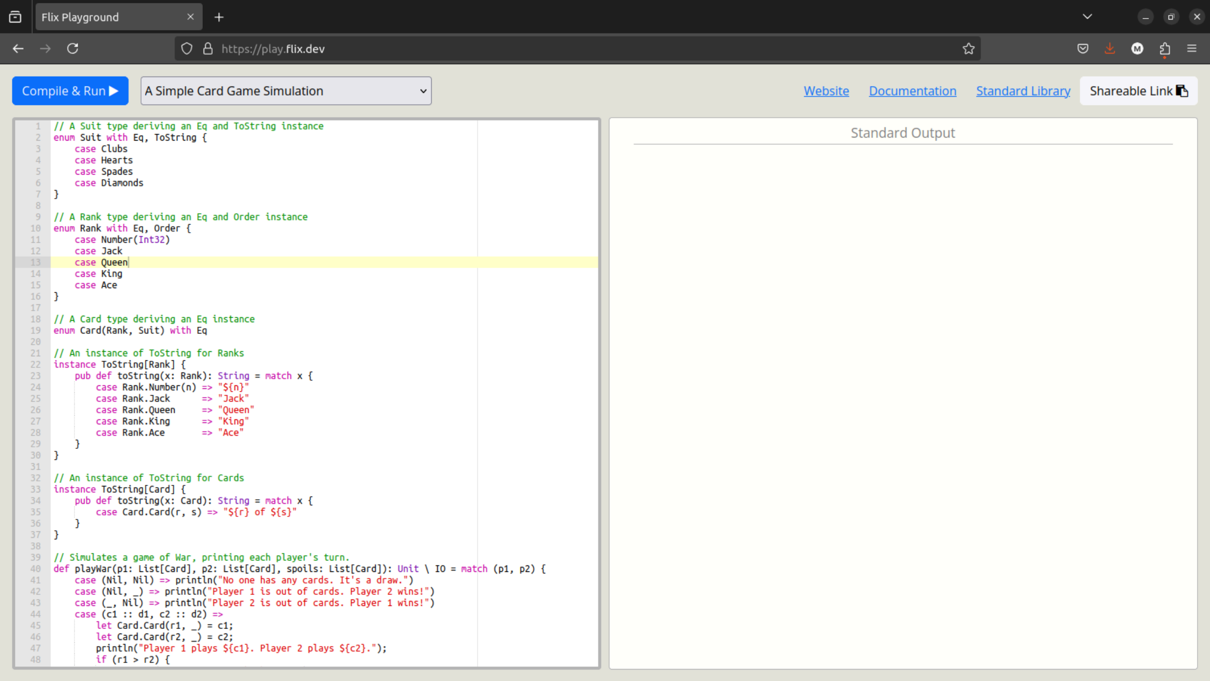
Moreover, Flix builds on proven programming language technology, including: **algebraic data types and pattern matching**, **extensible records**, **traits**, **higher-kinded types**, **associated types and effects**, **structured concurrency**, and more.

```
// Getting information on files with Flix.
def main(): Unit \ IO =
  let f = "README.md";

  // Check if the file 'README.md' exists.
  match File.exists(f) {
    case Ok(exist) => {
      println("The file ${f} exists: ${exist}.")
    }
    case Err(msg)  => println("An error occurred with message: ${msg}")
  };

  // Get statistics of the file 'README.md'.
  match File.stat(f) {
    case Ok(stats) => {
      println("${f} is of type: ${stats.fileType}");
      println("The size of ${f} is: ${stats.size}.");
      println("The creation time of ${f} is: ${stats.creationTime}.")
    }
    case Err(msg)    => println("An error occurred with message: ${msg}")
  }
```

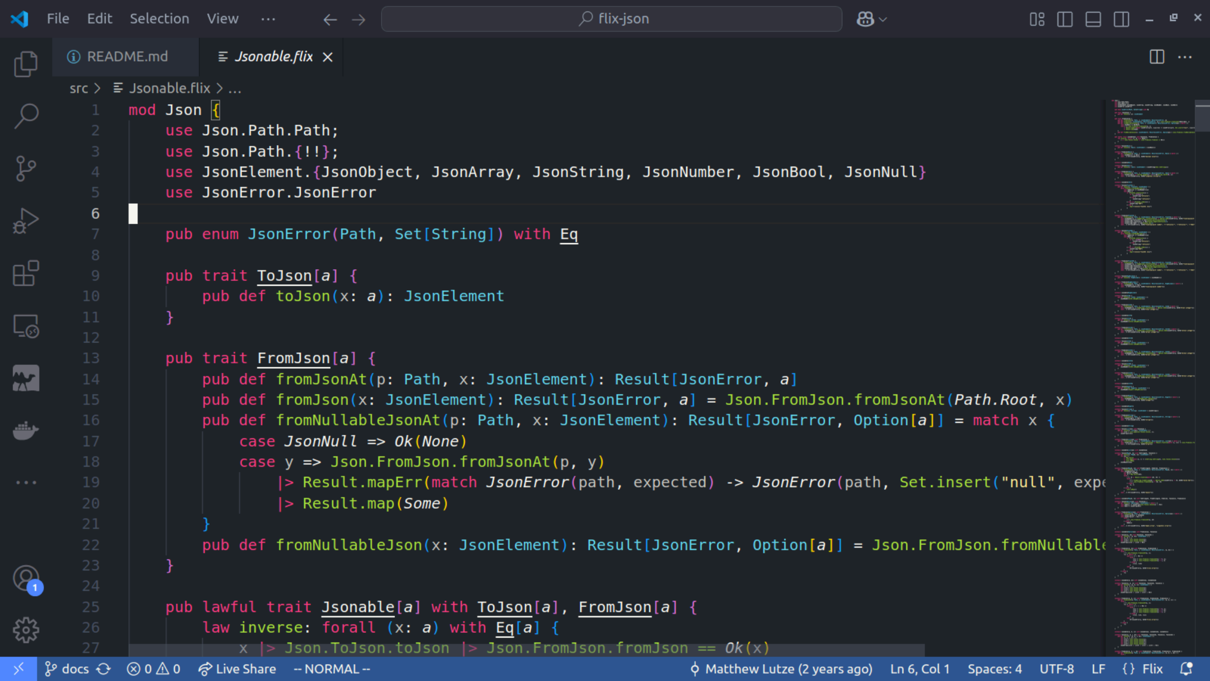
Algebraic Data Types and Pattern Matching

Algebraic data types and pattern matching are the bread-and-butter of functional programming and are supported by Flix with minimal fuss.

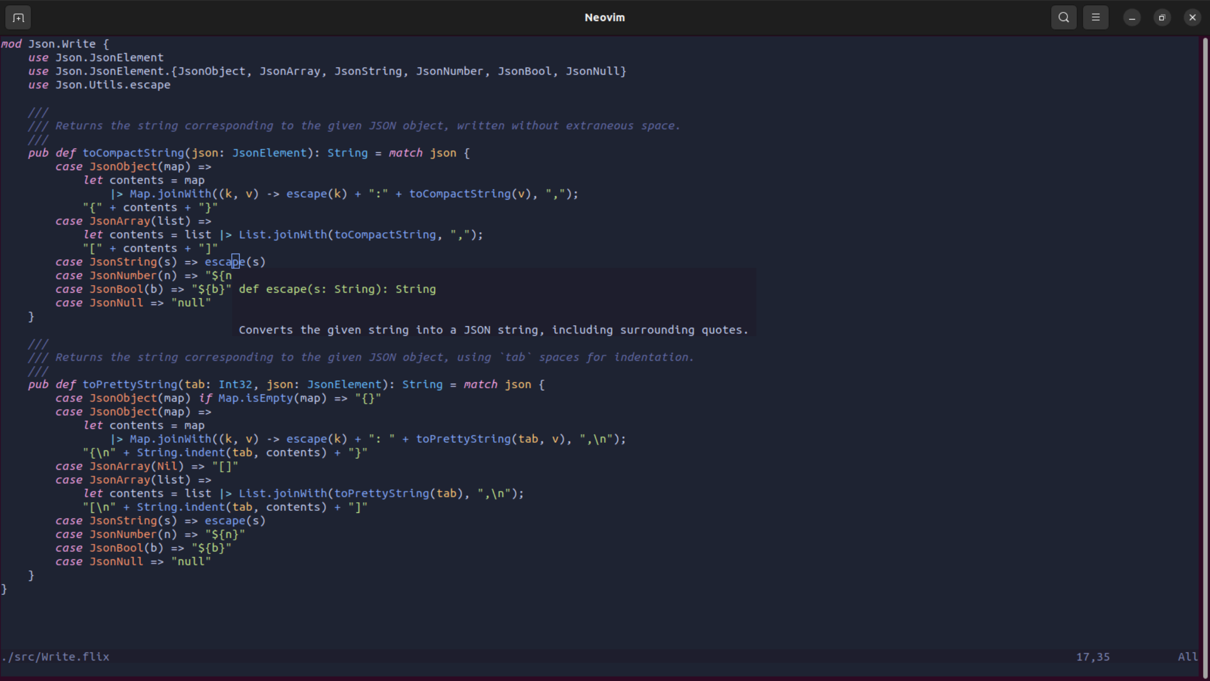
```
enum Shape {
    case Circle(Int32),
    case Square(Int32),
    case Rectangle(Int32, Int32)
}

def area(s: Shape): Int32 = match s {
    case Circle(r)       => 3 * (r * r)
    case Square(w)       => w * w
```



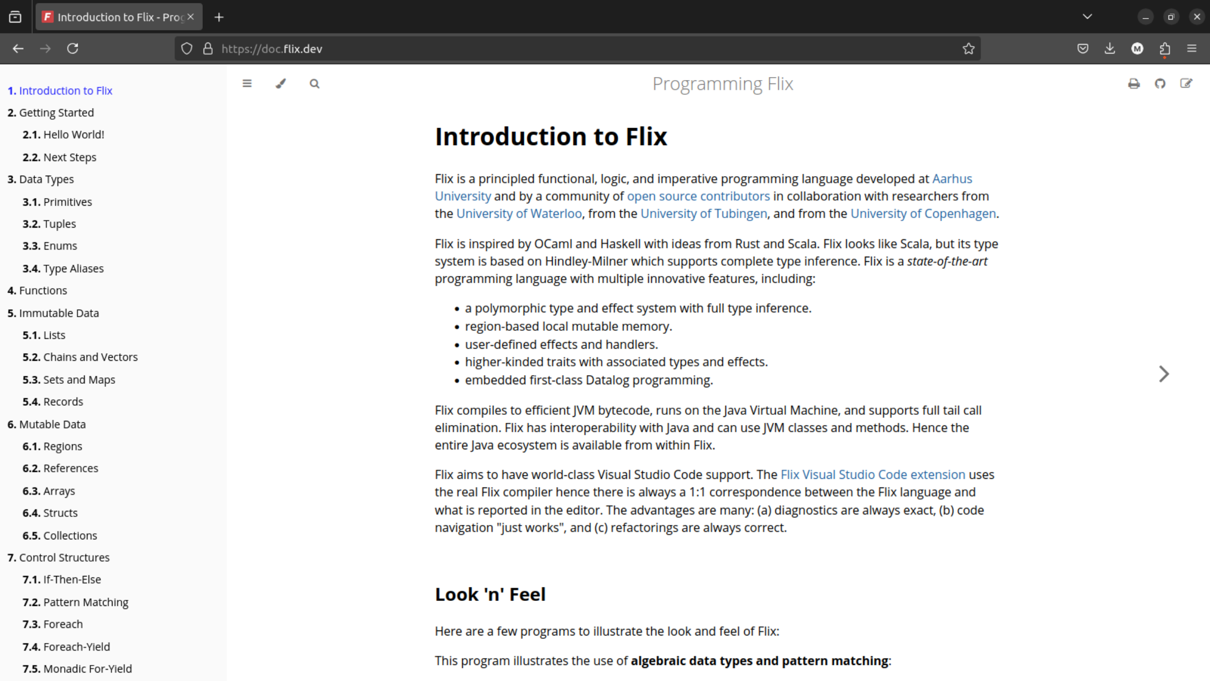
```
// A Suit type deriving an Eq and ToString instance
enum Suit with Eq, ToString {
    case Clubs
    case Hearts
    case Spades
    case Diamonds
}

// A Rank type deriving an Eq and Order instance
enum Rank with Eq, Order {
    case Number(Int32)
    case Jack
    case Queen
    case King
    case Ace
}

// A Card type deriving an Eq instance
enum Card(Rank, Suit) with Eq

// An instance of ToString for Ranks
instance ToString[Rank] {
    pub def toString(x: Rank): String = match x {
        case Rank.Number(n) => "${n}"
        case Rank.Jack      => "Jack"
        case Rank.Queen     => "Queen"
        case Rank.King      => "King"
        case Rank.Ace       => "Ace"
    }
}

// An instance of ToString for Cards
instance ToString[Card] {
    pub def toString(x: Card): String = match x {
        case Card.Card(r, s) => "${r} of ${s}"
    }
}

// Simulates a game of War, printing each player's turn.
def playWar(p1: List[Card], p2: List[Card], spoils: List[Card]): Unit \ IO = match (p1, p2) {
    case (Nil, Nil) => println("No one has any cards. It's a draw.")
    case (Nil, _) => println("Player 1 is out of cards. Player 2 wins!")
    case (_, Nil) => println("Player 2 is out of cards. Player 1 wins!")
    case (c1 :: d1, c2 :: d2) =>
        let Card.Card(r1, _) = c1;
        let Card.Card(r2, _) = c2;
        println("Player 1 plays ${c1}. Player 2 plays ${c2}.");
        if (r1 > r2) {
```



```
mod Json {
    use Json.Path.Path;
    use Json.Path.{!!};
    use JsonElement.{JsonObject, JsonArray, JsonString, JsonNumber, JsonBool, JsonNull}
    use JsonError.JsonError

    pub enum JsonError(Path, Set[String]) with Eq

    pub trait ToJson[a] {
        pub def toJson(x: a): JsonElement
    }

    pub trait FromJson[a] {
        pub def fromJsonAt(p: Path, x: JsonElement): Result[JsonError, a]
        pub def fromJson(x: JsonElement): Result[JsonError, a] = Json.FromJson.fromJsonAt(Path.Root, x)
        pub def fromNullableJsonAt(p: Path, x: JsonElement): Result[JsonError, Option[a]] = match x {
            case JsonNull => Ok(None)
            case y => Json.FromJson.fromJsonAt(p, y)
                |> Result.mapErr(match JsonError(path, expected) -> JsonError(path, Set.insert("null", expe
                |> Result.map(Some)
        }
        pub def fromNullableJson(x: JsonElement): Result[JsonError, Option[a]] = Json.FromJson.fromNullable
    }

    pub lawful trait Jsonable[a] with ToJson[a], FromJson[a] {
        law inverse: forall (x: a) with Eq[a] {
            x |> Json.ToJson.toJson |> Json.FromJson.fromJson == Ok(x)
```



```
mod Json.Write {
    use Json.JsonElement
    use Json.JsonElement.{JsonObject, JsonArray, JsonString, JsonNumber, JsonBool, JsonNull}
    use Json.Utils.escape

    ///
    /// Returns the string corresponding to the given JSON object, written without extraneous space.
    ///
    pub def toCompactString(json: JsonElement): String = match json {
        case JsonObject(map) =>
            let contents = map
                |> Map.joinWith((k, v) -> escape(k) + ":" + toCompactString(v), ",");
            "{" + contents + "}"
        case JsonArray(list) =>
            let contents = list |> List.joinWith(toCompactString, ",");
            "[" + contents + "]"
        case JsonString(s) => escape(s)
        case JsonNumber(n) => "${n
        case JsonBool(b) => "${b}" def escape(s: String): String
        case JsonNull => "null"
    }
                                   Converts the given string into a JSON string, including surrounding quotes.
    ///
    /// Returns the string corresponding to the given JSON object, using `tab` spaces for indentation.
    ///
    pub def toPrettyString(tab: Int32, json: JsonElement): String = match json {
        case JsonObject(map) if Map.isEmpty(map) => "{}"
        case JsonObject(map) =>
            let contents = map
                |> Map.joinWith((k, v) -> escape(k) + ": " + toPrettyString(tab, v), ",\n");
            "{\n" + String.indent(tab, contents) + "}"
        case JsonArray(Nil) => "[]"
        case JsonArray(list) =>
            let contents = list |> List.joinWith(toPrettyString(tab), ",\n");
            "[\n" + String.indent(tab, contents) + "]"
        case JsonString(s) => escape(s)
        case JsonNumber(n) => "${n}"
        case JsonBool(b) => "${b}"
        case JsonNull => "null"
    }
}

./src/Write.flix                                                                    17,35          All
```



1. Introduction to Flix
2. Getting Started
2.1. Hello World!
2.2. Next Steps
3. Data Types
3.1. Primitives
3.2. Tuples
3.3. Enums
3.4. Type Aliases
4. Functions
5. Immutable Data
5.1. Lists
5.2. Chains and Vectors
5.3. Sets and Maps
5.4. Records
6. Mutable Data
6.1. Regions
6.2. References
6.3. Arrays
6.4. Structs
6.5. Collections
7. Control Structures
7.1. If-Then-Else
7.2. Pattern Matching
7.3. Foreach
7.4. Foreach-Yield
7.5. Monadic For-Yield

Introduction to Flix

Flix is a principled functional, logic, and imperative programming language developed at Aarhus University and by a community of open source contributors in collaboration with researchers from the University of Waterloo, from the University of Tubingen, and from the University of Copenhagen.

Flix is inspired by OCaml and Haskell with ideas from Rust and Scala. Flix looks like Scala, but its type system is based on Hindley-Milner which supports complete type inference. Flix is a *state-of-the-art* programming language with multiple innovative features, including:

- a polymorphic type and effect system with full type inference.
- region-based local mutable memory.
- user-defined effects and handlers.
- higher-kinded traits with associated types and effects.
- embedded first-class Datalog programming.

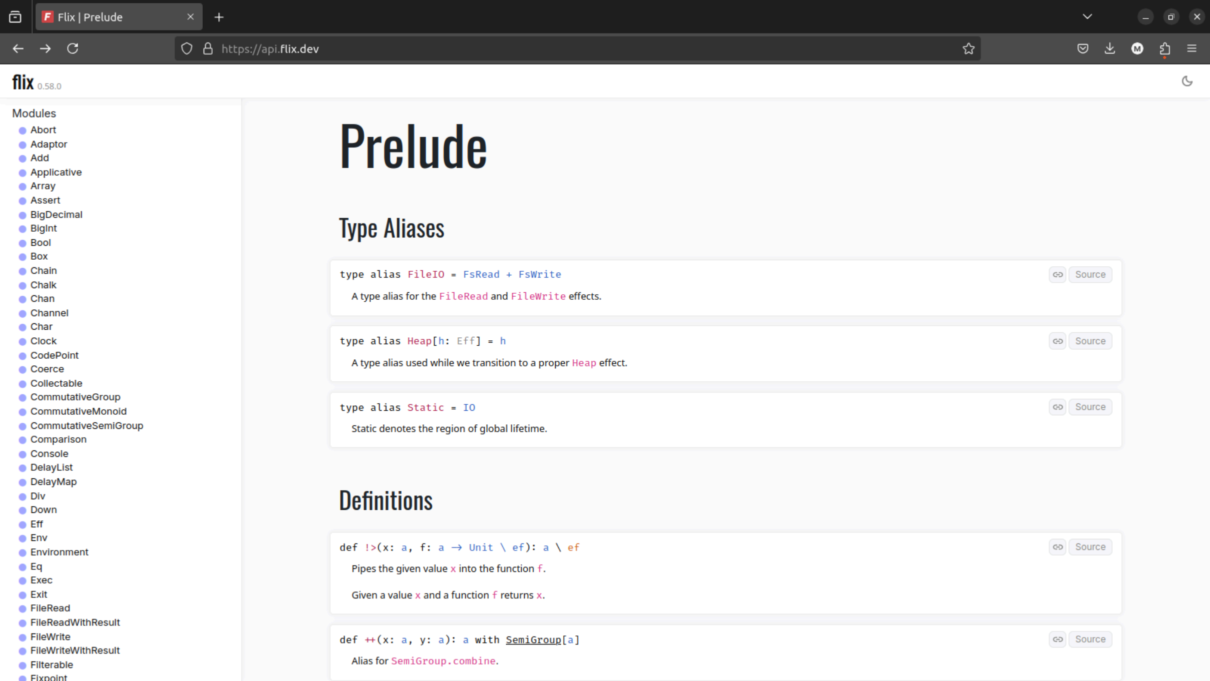
Flix compiles to efficient JVM bytecode, runs on the Java Virtual Machine, and supports full tail call elimination. Flix has interoperability with Java and can use JVM classes and methods. Hence the entire Java ecosystem is available from within Flix.

Flix aims to have world-class Visual Studio Code support. The Flix Visual Studio Code extension uses the real Flix compiler hence there is always a 1:1 correspondence between the Flix language and what is reported in the editor. The advantages are many: (a) diagnostics are always exact, (b) code navigation "just works", and (c) refactorings are always correct.

Look 'n' Feel

Here are a few programs to illustrate the look and feel of Flix:

This program illustrates the use of **algebraic data types and pattern matching**:



flix 0.58.0

Modules

- Abort
- Adaptor
- Add
- Applicative
- Array
- Assert
- BigDecimal
- BigInt
- Bool
- Box
- Chain
- Chalk
- Chan
- Channel
- Char
- Clock
- CodePoint
- Coerce
- Collectable
- CommutativeGroup
- CommutativeMonoid
- CommutativeSemiGroup
- Comparison
- Console
- DelayList
- DelayMap
- Div
- Down
- Eff
- Env
- Environment
- Eq
- Exec
- Exit
- FileRead
- FileReadWithResult
- FileWrite
- FileWriteWithResult
- Filterable
- Fixpoint

Prelude

Type Aliases

```
type alias FileIO = FsRead + FsWrite
```

A type alias for the `FileRead` and `FileWrite` effects.

```
type alias Heap[h: Eff] = h
```

A type alias used while we transition to a proper `Heap` effect.

```
type alias Static = IO
```

Static denotes the region of global lifetime.

Definitions

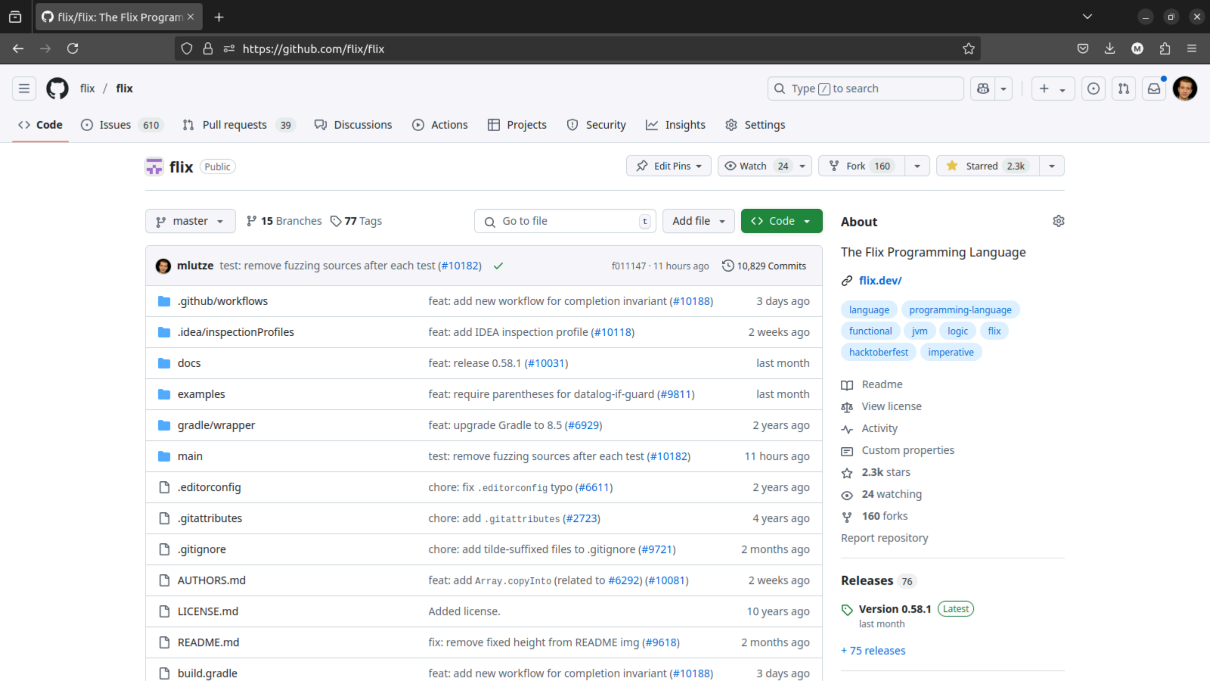
```
def !>(x: a, f: a → Unit \ ef): a \ ef
```

Pipes the given value `x` into the function `f`.

Given a value `x` and a function `f` returns `x`.

```
def ++(x: a, y: a): a with SemiGroup[a]
```

Alias for `SemiGroup.combine`.



mlutze test: remove fuzzing sources after each test (#10182) ✓ f011147 · 11 hours ago 10,829 Commits

.github/workflows	feat: add new workflow for completion invariant (#10188)	3 days ago
.idea/inspectionProfiles	feat: add IDEA inspection profile (#10118)	2 weeks ago
docs	feat: release 0.58.1 (#10031)	last month
examples	feat: require parentheses for datalog-if-guard (#9811)	last month
gradle/wrapper	feat: upgrade Gradle to 8.5 (#6929)	2 years ago
main	test: remove fuzzing sources after each test (#10182)	11 hours ago
.editorconfig	chore: fix `.editorconfig` typo (#6611)	2 years ago
.gitattributes	chore: add `.gitattributes` (#2723)	4 years ago
.gitignore	chore: add tilde-suffixed files to .gitignore (#9721)	2 months ago
AUTHORS.md	feat: add `Array.copyInto` (related to #6292) (#10081)	2 weeks ago
LICENSE.md	Added license.	10 years ago
README.md	fix: remove fixed height from README img (#9618)	2 months ago
build.gradle	feat: add new workflow for completion invariant (#10188)	3 days ago

About

The Flix Programming Language

flix.dev/

language programming-language functional jvm logic flix hacktoberfest imperative

Readme
View license
Activity
Custom properties
2.3k stars
24 watching
160 forks
Report repository

Releases 76

Version 0.58.1 Latest
last month

+ 75 releases

Visual Studio Code / LSP Support

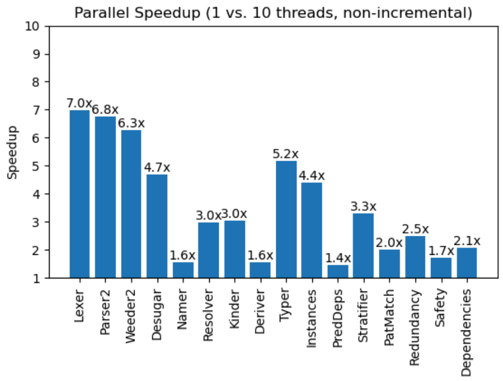
- ✓ syntax highlighting
- ✓ inline diagnostics
- ✓ auto-complete
- ✓ type and effect hover
- ✓ find references
- ✓ find implementations
- ✓ jump to definition
- ✓ code snippets
- ✓ automatic rename
- ✓ code hints
- ✓ code lenses
- ✓ document symbols
- ✓ workspace symbols
- ✓ highlight related symbols

Modern Compiler Architecture

Flix has a modern compiler which is **resilient**, **incremental**, and **parallel**.

Throughput	
frontend:	**140,382 lines/sec**
front + backend:	**60,159 lines/sec**

(On Apple M2 Pro with a 10-core CPU running on OpenJDK 21)



7 Wrapping Up

Project Contributors & Statistics

5,500+	Merged Pull Requests (PRs)
3,300+	Resolved Issues (Tickets)
70+	Contributors
250,000+	Lines in Compiler Codebase

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

UNIVERSITY OF
WATERLOO

Research Foundations (short excerpt)

Region-Based Memory Management [Inf. Comput. ‘97]

- *Mads Tofte, Jean-Pierre Talpin*

Handlers of Algebraic Effects [ESOP ‘09]

- *Gordon Plotkin, Matija Pretnar*

Koka: Programming with Row Polymorphic Effect Types [MSFP ‘14]

- *Daan Leijen*

Boolean Unification – The Story so Far [J. Sym. Comput. ‘89]

- *Ursula Martin, Tobias Nipkow*

How to make ad-hoc polymorphism less ad hoc [POPL ‘89]

- *Philip Wadler, Stephen Blot*

Associated Type Synonyms [ICFP ‘05]

- *Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones*

Foundations of Deductive Databases and Logic Programming [MKP ‘88]

- *Jack Minker et al.*

Selection of our Research

Associated Effects: Flexible Abstractions for Effectful Programming [PLDI '24]

- *Matthew Lutze, Magnus Madsen*

With or Without You: Programming with Effect Exclusion [ICFP '23]

- *Matthew Lutze, Magnus Madsen, Philipp Schuster, Jonathan Brachthäuser*

The Principles of the Flix Programming Language [ONWARD '22]

- *Magnus Madsen*

Polymorphic Types and Effects with Boolean Unification [OOPSLA '20]

- *Magnus Madsen, Jaco van de Pol*

Fixpoints for the Masses: Programming with First-Class Datalog Constraints [OOPSLA '20]

- *Magnus Madsen, Ondřej Lhoták*

Summary

Flix is a powerful **effect-oriented** programming language.

Flix aims to offer a **unique combination of features**:

Features

- algebraic data types and pattern matching
- traits with higher-kinded types
- a polymorphic type and effect system
- algebraic effects and handlers
- embedded Datalog
- Runs on the JVM

Tooling

- ✓ documentation and examples
- ✓ extensive standard library
- ✓ Visual Studio Code support
- ✓ generic LSP Support
- ✓ parallel and incremental compiler
- ✓ package manager

We are moving towards **version 1.0** and we want your feedback:

https://flix.dev/

Additional Resources

The Official Flix Website:	https://flix.dev
The Programming Flix Book	https://doc.flix.dev
API Documentation	https://api.flix.dev
Online Playground	https://play.flix.dev
GitHub	https://github.com/flix/flix
Twitter	https://twitter.com/flixlang
Gitter	https://gitter.im/flix/Lobby