The Principles of the Flix Programming Language

Magnus Madsen
magnusm@cs.au.dk
Department of Computer Science
Aarhus University
Aarhus, Denmark

Abstract

We present the design values and design principles of the Flix programming language, a functional-first, imperative, and logic programming language. We explain how these values and principles came into being and how they have influenced the design of Flix over the last several years.

The principles cover most facets of the Flix language and its ecosystem, including its syntax, semantics, static type and effect system, and standard library. We present each principle in detail, including its origin, rationale, and how it has shaped Flix.

We believe that codifying a language's design values and principles can serve as a powerful medium for discussing and comparing programming language designs and we hope our presentation will inspire future language designers to document their languages' design values and principles.

CCS Concepts: • **Software and its engineering → General programming languages**.

Keywords: Flix, programming language design, design values and principles

ACM Reference Format:
Magnus Madsen. 2022. The Principles of the Flix Programming Language. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '22), December 8–10, 2022, Auckland, New Zealand*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3563835.3567661

1 Introduction

Flix[1] is a functional-first, imperative, and logic programming language that supports algebraic data types, pattern matching, higher-order functions, parametric polymorphism, type classes, higher-kinded types, polymorphic effects, first-class Datalog constraints, channel and process-based concurrency, and tail call elimination. Flix compiles to JVM bytecode and runs on the Java Virtual Machine. Flix is developed by researchers at Aarhus University in collaboration with researchers at the University of Waterloo and Universität Tübingen, and by a community of open source contributors.

Flix has an extensive standard library and a rich ecosystem, including online documentation, API documentation, an online playground, and a fully-featured Visual Studio Code Extension. The Flix compiler project spans more than 190,000 lines of code written by more than 50 contributors.

In this paper, we discuss the *principled design* of the Flix programming language. In the programming language research literature, it is standard to present minimal calculi, including semantics and type systems, which focus on a particularly interesting aspect of a programming language. Various aspects of Flix has been described this way, including its support for first-class Datalog constraints [37, 40] and its polymorphic type and effect system [38]. But until now, there has not been a written account that tries to describe how the entire language is put together; *how it is designed.*

Every programming language designer faces a myriad of design choices. However, we argue that most of these choices are poorly understood and more rarely studied. Much research, including our own, has focused on "ensuring that well-typed programs do not go wrong" and less on "should unused local variables be allowed?" or "how does one design a good standard library?".

Programming language researchers and designers sometimes talk about a language being "opinionated" (or "unopinionated"). For example, Martin Odersky, the lead designer of Scala, has said that one of the goals of Scala 3 was for the language "to become more opinionated" [49]. We agree with this sentiment; programming languages should be opinionated.

But, what does it mean for a language to be opinionated? We think it means that it is the job of the language designer to make informed *choices* on behalf of the programmer. An opinionated designer should not throw every feature into a language and let the programmer sort it out. Instead, the language designer should establish a common way to structure and think about programs written in their language.

In this paper, we present a collection of values and principles that govern the design of the Flix. We do not claim that these values or principles are inherently good and bad;

[1] https://flix.dev/ and https://github.com/flix/flix/

Onward! '22, December 8–10, 2022, Auckland, New Zealand

ACM ISBN 978-1-4503-9909-8/22/12.
https://doi.org/10.1145/3563835.3567661

instead, we aim to use them to build a coherent, opinionated, and well-designed programming language. We think of values as aspirations toward some abstract goals and principles as actionable and decidable criteria for achieving those goals.

We believe that the idea of explicitly codifying the values and principles of a programming language may offer a more systematic method for the softer aspects of programming language design. In this methodology, we should not view the values and principles as set in stone but rather as a living document of design choices that are collected and published.

Today, it seems that few programming languages have anything resembling the values and principles outlined in this paper. We hope to be part of an impetus to change that. We can now state what we mean by *values* and *principles*:

Definition 1.1 (Value)**.** A *value*, in the broadest sense, is a subscription to a particular view on how programs should be written. A value, in a more concrete sense, is a property of a programming language that its designers consider to be important, worthy, or useful.

Definition 1.2 (Principle)**.** A *principle* is a design guideline; it expresses a particular property or collection of design choices about how something should work.

A principle should be *actionable* and *decidable*; i.e., it should be possible to look at a programming language and *determine* if it satisfies the principle.

To give an analogy: Imagine an organization with members. A specific value of the organization could be that "everyone should have their voice heard." A principle, inspired by that value, could be that "at every meeting, everyone gets to speak for two minutes". We may find it difficult to determine if the organization is living up to its values. But, it is much more straightforward to determine if it satisfies its principles and if *other* organizations satisfy the same principles.

Every principle comes with a rationale: an explanation of why the principle exists and is worthwhile. For some principles, the rationale is grounded in the research literature. For others, it comes from hard-earned lessons from other programming languages. Sometimes, a principle rests solely on a rhetorical argument or simply instinct. In these cases, we have tried to outline hypotheses that can serve as a guide for future empirical programming language research.

We like to think of the Flix values and principles as a social contract: A codification of what programmers can expect from us, the language designers, and what we, as language designers, expect from the programmers. We believe that by explicitly stating the Flix values and principles, programmers can more easily determine whether Flix is the right choice for their programming task. We also think that our values and principles are an effective way of communicating our perspectives on language design and to indicate in what direction Flix is headed.

The Flix values and principles came into being over several years. In the beginning, they were informal and not written down. Later, we started writing them down and publishing them on the Flix website to keep us honest and solicit feedback. Over these years, the principles arose out of: (i) discussions on GitHub, (ii) discussions on other programming language forums, and (iii) from the perceived mistakes of other programming languages. Later, we came to an informal process where when a design discussion ended in agreement, we would ask ourselves: is there a principle here that should be codified and put on the website? This paper is the culmination of that process; it is not merely the outcome of several weeks of writing but the result of six years of discussions on programming language design.

The Flix values and principles are essential to consensus-building among the Flix developers. It is much easier to collaborate when there is a shared written agreement on many aspects of the language design. For example, in our experience, principles help reduce tension during code review because reviewers can offer constructive criticism by pointing to pre-established principles. When no principle exists, yet the language designers agree that a pull request or design choice is "wrong", tensions can be defused by saying that *there should have been such a principle* and discussing it with the contributor. Thus the designers have a chance to act less autocratic, and the contributor is included in a positive way that does not attack his or her work.

Flix has often been discussed on the internet with multiple posts on websites such as HackerNews, Reddit, and Lambda the Ultimate. Surprisingly – at least to us – these discussions have often focused on the Flix values and principles rather than the technical merits of the language or the scientific contributions of Flix. We speculate that this is because the values and principles communicate what Flix is and aspires to be, which facilitates more productive discussions on programming language design.

As of today, Flix has more than 60 values and principles[2]. We cannot cover all of them in this paper, hence we focus on the ones we think are the most important or have had the biggest impact on the language.

We organize the principles into the following categories: *Syntax* (Section 4.1), *Static Semantics* (Section 4.2), *Correctness and Safety* (Section 4.3), *Compiler Messages* (Section 4.4), *Standard Library* (Section 4.5), *Miscellaneous* (Section 4.6), and finally *Abandoned Principles* (Section 5).

This paper contains more than rhetorical arguments; we present the values and principles and discuss their success and failure in shaping Flix. It is our goal to illustrate how each principle has influenced the design of Flix. We also discuss some surprising results. For example, principles that were thought straightforward but applying them uncritically led to non-sensical design choices. As often said, the devil is in the details. Lastly, we also discuss some principles that we had to modify or abandon over the years.

[2] https://flix.dev/principles/

2 Background: A Taste of Flix

We begin with a brief introduction to Flix. Unfortunately, we cannot cover the language in great detail, but we want to give the reader a taste of the look and feel of the language.

2.1 A Brief Introduction to Flix

Flix is a functional-first, imperative, and logic programming language. We illustrate each style of programming:

Functional-style. We can write programs in a functional style using algebraic data types and pattern matching:

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

def area(s: Shape): Int32 = match s {
    case Circle(r)       => 3 * (r * r)
    case Rectangle(h, w) => h * w
}
```

Here we define an algebraic data type named `Shape` with two variants: `Circle`, and `Rectangle`, and a function `area` to compute the area of a given shape.

Imperative-style. We can also write programs in an imperative style using mutable data and destructive operations:

```
def sort(l: List[a]): List[a] with Order[a] =
    region r {
        let arr = List.toArray(l, r);
        Array.sort!(arr);
        Array.toList(arr)
    }
```

Here we implement the *pure* `sort` function by declaring a new lexically scoped region `r`, converting the immutable linked list `l` to a mutable array which belongs to that region, destructively sorting the array in place, and converting it back to an immutable linked list.

Declarative-style. And finally we can write programs in a declarative style using first-class Datalog constraints:

```
def reach(g: List[(t, t)]): List[(t, t)] =
    let db = inject g into Edge;
    let pr = #{
        Path(x, y) :- Edge(x, y).
        Path(x, z) :- Path(x, y), Edge(y, z).
    };
    query db, pr select (x, y) from Path(x, y)
```

Here the `reach` function computes the transitive closure of the graph `g` using Datalog.

2.2 Type and Effect System

A unique feature of Flix is its polymorphic type and effect system which separates pure and impure code [38]. In Flix, we can express that a function is pure (i.e., has no side-effects):

```
def add(x: Int32, y: Int32): Int32 \ { }
```

We can also express that a function is impure:

```
def sayHello(name: String): Unit \ { IO } =
    println("Hello ${name}!")
```

And we can express that the effect of a higher-order function depends on its argument (i.e., it is effect polymorphic):

```
def map(f: a -> b \ e, l: List[a]): List[b] \ e
```

Here the purity of the map function depends on the purity of its function argument f.

2.3 Ecosystem and Tooling

Flix has a website that describes the language (flix.dev), online documentation (doc.flix.dev), API documentation à la Javadoc (api.flix.dev), an online playground (play.flix.dev), and a fully-featured Visual Studio Code extension.

Flix comes with an extensive standard library that includes common functional data structures such as Option, Result, List, Set, and Map. The library also includes common type classes, such as Eq, Order, Functors, Monads, Foldables, and their standard instances. The Flix library spans more than 30,000 lines of code and offers more than 2,600 functions. Flix also has a Java FFI making it possible to reuse large parts of the Java Class Library.

Flix is ready for use, open source, and freely available at:

https://flix.dev/ and https://github.com/flix/flix/

2.4 A Brief History of Flix

Flix started as a functional and logic programming language, essentially a Datalog dialect enriched with lattice semantics [36, 40, 41]. Rather quickly, Flix turned into a full-blown functional programming language, and Flix swallowed the logic programming language by allowing Datalog constraints as first-class values [37]. Next, a polymorphic type and effect system, based on Boolean unification, was added to separate pure and impure code [38]. Around the same time, higher-kinded types and type classes were added to the language. Recently, the effect system has been extended with regions allowing a new style of functional imperative programming.

Today, six years later, development on Flix is fast-paced. We recently passed 4,000 issues and pull requests on GitHub. The Flix compiler project spans more than 190,000 lines of code with contributions from more than 50 developers.

3 The Flix Values

We anchor the Flix design principles in a collection of *values*. Each value is a *quality* or *trait* we, as language designers, find valuable and want Flix to have.

We now discuss the most important Flix design values:

Value 1 (**Simple is Not Easy**). *Flix adopts Rich Hickey's maxim: "simple is not easy" [51]. That is, we prefer design choices that get things right over ones that make things easy.*

Rationale. Rich Hickey's maxim is really about necessary complexity: whether to hide it (and hope for the best) or whether to surface it (and force the user to deal with it). What Hickey expresses, which we agree with, is that it is fundamentally necessary to deal with inherent complexity. For example, many programming languages have a null value. If null is a member of every type then it is *easy* to represent absent or uninitialized values, but it also makes it incredible difficult, i.e., not *simple*, to understand when a value is guaranteed to be non-null.

Value 2 (**Principle of Least Surprise**). *Flix should use safe and predictable defaults. When no reasonable default is apparent, there should not be a default; instead the programmer should make an explicit choice.*

Rationale. This value, which we can describe as "explicit is better than implicit", says that a programming language should never surprise the programmer. We give two counter-examples: First, JavaScript supports implicit coercions that silently convert one type into another following a complex collection of rules. Programmers often struggle to understand these rules leading to many bugs. Second, in Java every class implicitly implements the equals and hashCode methods even though not all objects may have a meaningful equality relation (e.g. objects that represent closures).

Value 3 (**Functional-First**). *Flix is a functional, imperative, and logic programming language. But, Flix is foremost a functional language, and if there is a conflict between these paradigms, then it should be resolved in favor of the functional aspects of the language.*

Rationale. There is an inherent tension between functional– and imperative programming. The value states that when a trade-off is made, it should never be at the expense of the functional language. For example, we have already seen that Flix uses its type and effect system to separate pure and impure code. Thus a functional programmer does not lose the benefits of equational reasoning when writing pure code.

Value 4 (**One Language**). *Flix is one programming language. Flix does not have feature flags, compiler plugins, or any other means of changing the language.*

Rationale. We want to avoid fragmentation of the nascent Flix ecosystem. We do not want programs written in different dialects of Flix. There should be one Flix language. This is about control; we should always have a clear mental model of what is and what is not in the language.

Value 5 (**Productivity Over Performance**). *Flix values developer productivity over runtime performance. If there is trade-off between development speed and runtime performance, then it should resolved in favor of development speed.*

Rationale. Flix aims to support developer productivity; the ability to get a lot done with little ceremony and boilerplate. The fundamental value is that development time is a more expensive resource than hardware. A carefully crafted C program might run faster than a Flix program by default, but it will not be as short, concise, or quick to write as the Flix program. Flix aims to be a language with powerful constructs and high-level abstractions. In Flix, programmers should be able to write programs quickly, even if it may come at the cost of performance. For example, functional data structures are simpler to reason about than their imperative counterparts but are often a logarithmic factor slower. A programmer can still write blazingly fast Flix code (and the compiler still optimizes code), but programmers should not be forced to deal with performance aspects unless they choose to.

Value 6 (**Correctness Over Performance**). *Flix values program correctness over performance. If there is a trade-off between correctness and performance, then it should be resolved in favor of correctness.*

Rationale. Low-level programming languages like C and C++ often rely on undefined behavior to achieve stellar performance. In contrast, higher-level languages like Haskell, Java, and JavaScript try to avoid undefined behavior. Both Java and Flix benefit from being hosted on the Java Virtual Machine (JVM) which ensures memory safety. For example, the JVM performs dynamic checks to ensure that every array access is within bounds and safely aborts the execution if an access is out-of-bounds. However, Flix goes further. For example, Flix supports full tail call elimination[3] to ensure that tail calls never overflow the stack. Since the JVM lacks native tail calls, Flix must emulate tail calls using trampolines. This is an example where Flix trades performance for correctness: A Flix programmer can safely write programs using tail calls without worrying of the program crashing, but it comes at a (small) price.

[3]Note that full tail call elimination [42, 53] is more powerful than simple tail recursion optimization. In Flix, any call that is in tail position is guaranteed not to increase the stack height. Thus, a program in CPS can be evaluated without blowing the stack, which is not necessarily the case in other functional programming languages on the JVM (e.g. Scala).

4 The Flix Principles

We now discuss the Flix design principles. A principle, unlike a value, is a particular property of a programming language design that is actionable and decidable, i.e., we should be able to enforce it, and we should be able to assess whether a language satisfies a specific principle.

4.1 Syntax Principles

Principle 1 (Syntax vs. Semantics). *Syntax and semantics are essential, but the two should not be confused.*

Rationale. The essence of this principle is to avoid confusion between syntactic and semantic issues: a syntactic issue should not be solved with enrichment of semantics. We give two counter-examples: extension methods [59] and implicit classes [52]. Both techniques allow programmers to enrich classes with new methods. For example, the String class does not have a center method, but programmers can use extension methods (or implicit classes) to add such a method retroactively. Thus programmers can then write `s.center()`, as if center was a method on String. The problem is that extension methods and implicit classes come with a lot of semantic baggage. For example, in Scala 2.x, using an implicit class has the downside that, in certain circumstances, the compiler cannot eliminate the construction of the implicit object. Thus the ability to write `s.center()` instead of `center(s)` may incur an unexpected performance penalty. In summary, the purpose of this principle is to encourage a strong separation between syntactic issues (which may be solved by syntactic sugar) and semantic issues (which may be solved by language extensions).

Discussion. As a functional language, Flix does not have traditional for- or while-loops. However, we recently added syntactic sugar for two kinds of "for-loops"; one is a foreach-loop that is syntactic sugar for using an Iterable, and the other is a monadic-do that is syntactic sugar for using the map and flatMap functions. Importantly these additions are purely syntactic and *do not* modify the semantics of Flix.

Principle 2 (Expression-based Syntax). *Flix is primarily a functional language and embraces the idea that every construct should be an expression.*

Rationale. The idea is that every programming construct should reduce to a value that can be used for further computation. For example, Flix has no local variable *declarations* or if-then-else *statements*, instead it has let-bindings and if-then-else *expressions*.

Discussion. Flix does not take this idea as far as the Scheme languages where every construct is an expression. Flix still has namespaces declarations and type declarations, neither of which are expressions. Flix also has statement expressions: `e1; e2` for working with expressions that are executed purely for their side-effects.

Principle 3 (Keyword-based Syntax). *The Flix syntax is based on keywords. A keyword should be short, ideally three letters, and visually help the programmer understand the structure of a program.*

Rationale. We make the following observations:

- We conjecture that keywords help programmers read and understand the structure of code better than just symbols and punctuation.
- Keywords enable basic syntax highlighting in most IDEs and editors.
- The length of a keyword is essential to ensure a pleasing visual alignment. Flix uses 4-space indentation, which works well with three-letter keywords since three letters and a space aligns with 4-spaces.

Discussion. Early versions of Flix used `!`, `&&`, and `||` as the syntax for the standard Boolean operators. Inspired by this principle, these operators were changed to the keywords: `not`, `and`, and `or`. This design choice was supported by two additional observations: First, a single `!` symbol is easy to overlook while scanning through or reading source code. Second, unlike all other operators, the semantics of `and` and `or` short-circuit. We felt that turning these operators into keywords would also help highlight this aspect.

Hypothesis. We propose the hypothesis that keywords help programmers understand the structure of their code better than symbols and punctuation.

Principle 4 (Mirrored Term and Type Syntax). *Flix should have consistent term and type-level syntax.*

Rationale. A Flix programmer should not have to learn and remember two different syntaxes for every construct. Instead, the syntax of an expression and the syntax of its corresponding type should be similar. For example:

- A function application is written as `f(a, b, c)` whereas a type application is written as `f[a, b, c]`.
- A function expression is written as `x -> x + 1` whereas a function type is written as `Int32 -> Int32`.
- A tuple expression is written as `(true, 12345)` whereas a tuple type is written as `(Bool, Int32)`.
- A record expression is written as `{ x = 123 }` whereas a record type is written as `{ x = Int32 }`.
- A Datalog expression is written as `#{ A(123). }` whereas a Datalog type is written as `#{ A(Int32) }`.

Discussion. While revisiting this principle, we discovered that the syntax for records and record types was inconsistent. Without any particular reason, we had adopted the syntax `{ x :: Int32 }` for records types. This has now been corrected.

Flix has infix function applications which allow a function call `sum(x, y)` to be written as `x 'sum' y`. We experimented with allowing the same for types, i.e., `Map[k, v]` could be written as `k 'Map' v`, but ultimately we found this too strange and unfamiliar, and it was removed.

4.2 Static Semantic Principles

Principle 5 (Separate Pure and Impure Code). *Flix should separate pure and impure code. A programmer should know when a function behaves as a mathematical function, i.e., returns the same output when given the same input(s) without causing any side effects.*

Rationale. An essential property of functional programming is *equational reasoning*, i.e., the ability to reason about functions based solely on their inputs. However, in a hybrid programming language with side effects, such reasoning is often lost. This principle, a cornerstone of the Flix language, states that it should be possible to write functional code in Flix *and* know that it supports equation reasoning.

Flix uses a powerful type and effect system to precisely track the purity or impurity of every expression [38]. The effect system enables programmers to enforce purity and to know *when* a function behaves as a mathematical function.

Principle 6 (Separate Pure and Impure Data). *Flix should separate pure and impure data; i.e., data that is immutable from data that is mutable.*

Rationale. This principle, which is related to the previous, states that all data should be divided into two categories: immutable and mutable. Immutable data is unchanging and functions that only operate immutable data, and which do not access the outside world, are guaranteed to be pure. Moreover, immutable data can freely be sent between threads, whereas mutable data must never be shared to prevent data races and race conditions. In Flix, all mutable data belongs to a region and cannot escape the lifetime of the region. When the region goes out of scope, any effects associated with the region also vanish. Thus, pure functions can be implemented using local mutable data, as shown in Section 2.

Principle 7 (Complete Local Type Inference). *Flix should be able to infer all types and effects within a function while only requiring type signatures for top-level declarations.*

Rationale. We believe that the rise of untyped programming languages, such as Python and JavaScript, demonstrates that type inference is essential for statically typed programming languages to become successful. Consequently, Flix aims to support local, but infallible, type inference.

In Flix, all top-level functions must have type signatures, but within a function no type annotations are ever required; not for nested functions nor for lambdas. This has three distinct advantages:

- Type signatures are useful as documentation.
- Type signatures accurately assign blame.
- Type signatures enable parallel type inference.

Discussion. Broadly speaking, there are two classes of type systems that support type inference: bi-directional type systems [15, 50] and Hindley-Milner-style systems [9, 22, 46, 60]. A bi-directional type system supports type inference but may require annotations on function arguments (or risk incompleteness). In contrast, a Hindley-Milner-style type system has complete type inference but is typically less expressive than a bi-directional system. The Flix type and effect system is based on Hindley-Milner and the papers that describe it all use Hindley-Milner as the formal framework [38, 39]. This is reasonable since Hindley-Milner guarantees completeness of type inference. However, Flix does not fully take advantage of Hindley-Milner because of the requirement that top-level declarations must have type signatures. In our experience, this design is very effective: Programmers write top-level signatures, as they would also do in Kotlin, Java, Scala, and many other programming languages, but within a function, type inference is infallible and cannot give spurious type errors like in the aforementioned languages.

Principle 8 (Whole-Program Compilation). *Flix requires all code to be available at compile-time.*

Rationale. The requirement that the compiler must have access to the entire source code has several advantages, it:

- Enables monomorphization which increases inlining opportunities and avoids the need to box primitives.
- Enables aggressive dead code elimination and tree shaking which significantly reduces code size.
- Enables cross namespace/module optimizations.
- Ensures global uniqueness of type class instances [5].

In the past, requiring access to the entire source code of a program may been impractical. Today, many programming languages, including JavaScript, Python, PHP, and Rust have ecosystems where packages include the entire source code.

Principle 9 (Single Entry Point). *A Flix program has a single entry point and no code is executed before it.*

Rationale. Dijkstra, in his "Notes on Structured Programming", proposed that procedures should only have a single entry- and single exit point [13]. In a similar spirit, this principle states that an entire Flix program should have one entry point: `main`. The rationale is simple: to enable programmers to understand and control the execution flow from start to finish. Today, this is not the case in many programming languages. For example, in many scripting languages, like JavaScript and Python, a program is just a script, and it may be challenging to identify where the main entry point is. Even in C# and Java, which have a designated `main` method, it is not the first thing to be executed. For example, in Java, static initializers, of all loaded classes, are executed before entering `main`. Worse, such initializers are allowed to have arbitrary side-effects. The existence of class loaders and over-reliance on reflection only exacerbates this problem. Thus, Flix enforces that `main` is the single entry-point and that it is the first and only thing executed.

Principle 10 (Minimize Declarations). *Flix should require as few declarations as possible.*

Rationale. Flix is statically typed but wants to emulate the flexibility and ease of use of dynamically typed languages. We believe one way to achieve that is by using structural typing combined with complete type inference. Flix uses structural types for tuples and extensible records [31], but also for first-class Datalog values [37].

Discussion. Early versions of Flix also supported first-class Datalog programs but required every predicate symbol and the types of its terms to be explicitly and globally declared. This design was inflexible and verbose, but fortunately, we were able to switch to a structurally typed system later.

Principle 11 (Private by Default). *In Flix, declarations are private by default and cannot be accessed from outside their namespace. A declaration must be explicitly marked as public to be visible to the outside world.*

Rationale. Flix embraces the principle of least privilege. We believe that programmers must make a conscious choice before a program construct is made publicly available, and other programmers start depending on it.

Principle 12 (Timeless Design). *The Flix compiler and standard library should not add support for a technology before it has matured and established its permanency.*

Rationale. If Flix had been designed two decades ago, adding language support for HTML and XML would have been tempting. If Flix had been designed one decade ago, adding language support for JSON would have been tempting. If Flix had been designed five years ago, adding language support for YAML would have been tempting. We are not saying these technologies are necessarily dated or bad, but just observing that their popularity waxes and wanes. Today, most programmers would probably be surprised and dissatisfied if one designed a new programming language with comments expressed in XML. This principle exists to ensure that the Flix language and standard library remain timeless.

4.3 Correctness and Safety Principles

Principle 13 (No Warnings, Only Errors). *Flix should never emit warnings; only errors that abort compilation.*

Rationale. Warnings occupy a gray zone where software developers may not fully understand whether they are harmless and can be safely ignored or if they are dangerous and should be fixed. Worse, software companies and developers may have different standards; some may ignore all warnings, while others may turn all warnings into hard errors. We believe that it is the responsibility of the language designer to determine if something is harmful and should be banned; or if something is safe and should be permitted.

Principle 14 (No Global State). *Flix has no global shared state; there are no global constants or static fields. The only global state is the external environment, i.e., the operating system, the file system, and so on.*

Rationale. Global state is the source of a plethora of issues, including its anti-modular nature, issues with initialization, and the threat of data races or race conditions in a multi-threaded environment. A Flix programmer can still emulate global state; he or she can create mutable memory in main and pass it around, but must do so explicitly.

Discussion. Global state is, as the principle states, impossible to avoid: the outside world is stateful. The operating system, file system, network, remote services, and so forth are all stateful. But, Flix programmers and the Flix compiler can use the type and effect system to know when a computation depends on the external environment, i.e., is impure.

Principle 15 (Share Memory by Communicating). *Flix follows the Go mantra: "Do not communicate by sharing memory; instead, share memory by communicating" [2]. In other words: communicate using immutable messages, not by sharing mutable memory.*

Rationale. The Flix concurrency model is based on channels and light-weight processes [23]. As discussed earlier, Flix separates immutable and mutable data and uses the type and effect system to enforce that only immutable values can be sent over channels, i.e., between processes. This approach, although heavy-handed, ensures the absence of data races, except on channels and through interoperability with Java. In the future, it may be possible to reduce this requirement by exploiting regions to ensure that mutable memory is never accessed by more than one thread.

Principle 16 (Concurrency vs. Parallism). *Flix should support both concurrency and parallelism but not confuse the two. In particular, it should be possible to write parallel programs without the inherent dangers of concurrency.*

Rationale. Roughly speaking, concurrency is the ability for two or more tasks to start, run, and complete in overlapping time-periods, whereas parallelism is the ability for two or more tasks to execute simultaneously. Concurrency can lead to dangers due to unexpected thread interleavings (e.g. race conditions) which can be prevented with proper usage of locks, but these can lead to deadlocks. This principle captures that Flix should support (safe) parallelism allowing programmers to execute two or more *independent* tasks simultaneously.

Discussion. Flix supports safe parallelism via two constructs: `par–` and `par-yield`-expressions. The `par (exp1, exp2, exp3)` expression evaluates exp_1, exp_2, and exp_3 in parallel to three values v_1, v_2, and v_3, and returns the tuple (v_1, v_2, v_3). The Flix type and effect system enforces that the expressions must be pure, hence they can be evaluated completely independently.

The `par-yield` construct is similar, but allows the programmer to bind the results to variables:

```
par (x <- exp1, y <- exp2, z <- exp3)
    yield x + y + z
```

Here the expressions exp_1, exp_2, and exp_3 are evaluated in parallel, their results bound to the variables x, y, and z, and then the body expression is evaluated.

Principle 17 (Bugs are Not Recoverable). *Flix follows the Midori Error Model [14] which states that there are two types of errors: (a) recoverable errors, and (b) program bugs. Recoverable errors are things like illegal user input, network errors, etc. Errors that can be anticipated and where there is a chance of recovery. Program bugs, on the other hand, are always unexpected and cannot be recovered.*

Rationale. Inspired by both the Midori Error Model and the Rust standard library, the Flix standard library goes to great lengths to model every fallible operation as returning a value of the Result data type, i.e., either Ok(v) or Err(v). Thus the programmer is forced to consider the possibility of failure. For example, the File.readLines function has the return type Result[List[String], IOError] which forces the programmer to handle both the happy path and the error path.

Discussion. Flix has been designed to minimize the way a program can go wrong. For example, Flix defines division by zero as returning zero [11]. Nevertheless, there are a few ways a Flix program can crash:

- by running out of resources, e.g., stack or heap space.
- by indexing an array with an out-of-bounds index.
- by calling Java code that crashes.

For completeness, we also mention that a Flix program can deadlock by waiting forever on a channel, e.g., waiting for a message to arrive or to be sent.

Principle 18 (No Useless Expressions). *Flix rejects programs that contain expressions that have no effect.*

Rationale. Redundant or dead code is a code smell and is correlated with program bugs [61]. Flix uses its powerful type and effect system to reject such useless expressions. For example, in the expression statement: `e1; e2`, Flix requires that `e1` must have a side-effect, i.e., if `e1` is pure then the program is rejected. Moreover, `e1` must also have the Unit type. If not, the programmer must insert an explicit `discard` expression to inform the compiler that it is okay to throw away the non-Unit value of `e1`.

We believe such simple checks may help programmers avoid trivial mistakes. For example, a programmer might write: `checkPermission(...); sensitiveOperation(...)` expecting that the call to checkPermission will ensure that we do not call sensitiveOperation unless we have permission. But it is possible that checkPermission returns a Bool that the programmer is supposed to inspect. Flix helps catch and prevent such mistakes.

Principle 19 (No Unused Variables). *Flix rejects programs with unused local variables, whether they are introduced by let-bindings, pattern matching, or as the formal parameters of a function.*

Rationale. While unused local variables are common during the programming process, their existence in finished code is a code smell. Moreover, research has shown that minor mistakes are a common source of bugs, e.g., using the wrong local variable. Disallowing unused local variables helps avoid such mistakes [26, 61].

Discussion. As a practical convenience, Flix allows any variable to be prefixed by an underscore to mark it as unused. This removes the variable from its scope and allows the program to compile and run.

Principle 20 (No Variable Shadowing). *Flix rejects programs with variable shadowing.*

Rationale. The rationale is the same as for unused variables; local variable shadowing is a common source of bugs.

Principle 21 (No Unused Declarations). *Flix rejects programs with unused declarations.*

Rationale. While unused declarations are less likely to be a source of bugs, they are nevertheless a code smell. By rejecting programs that contain unused declarations (functions, types, type aliases, etcetera), Flix ensures that all source code is actively in use.

Principle 22 (No Implicit Coercions). *Flix never coerces a value of one type into a value of another type.*

Rationale. Programming languages with implicit type coercions allow programmers to provide values of one type where values of another type are expected. The compiler or runtime automatically inserts code that re-interprets one value as another, often according to a sophisticated set of rules. For example, integer values may be promoted or truncated. Likewise, strings may be coerced to integers and vice versa. In some languages, all values may be coerced to Booleans. Unfortunately, such implicit coercions are brittle and error-prone. In some cases, information may be lost (e.g., when an integer is truncated), and in other cases, coercions may lead to unexpected results. JavaScript is notorious for its byzantine coercion rules and developers often use non-coercive operators to avoid such pitfalls. For these reasons, Flix never performs any coercions, not even information preserving promotions. Instead, programmers must manually and explicitly convert between types.

Discussion. While the Flix language does not support implicit coercions, a form of structured and predictable "implicit coercions" has snuck in using the backdoor. We might expect the println function to have the signature:

```
def println(s: String): Unit
```

but in the Flix standard library it actually has the signature:

```
def println(x: t): Unit with ToString[t]
```

which means that println can be called with any value of type t as long as t implements the ToString type class. This effectively works as an implicit coercion, except that:

- it is a very specific type of coercion,
- it is implemented using the type class system,
- it is clear from the signature of println that it makes use of this form of coercion, and finally
- the coercion rules are defined by the programmer as instances of the ToString type class.

Principle 23 (Declaration Monotonicity). *Adding a new declaration to a program should neither change its original semantics nor render it illegal.*

Rationale. It should always be safe to *extend* a program by adding new declarations. For example, adding a new function to a program should not make the original program behave differently nor should it prevent the program from compiling. (Of course, not every addition is safe; re-declaring the same function has to be a compile-time error.)

Discussion. This principle is the reason why Flix does not support wildcard imports. Imagine that we wrote:

```
use Foo.*
use Bar.*
qux()
```

where the namespace Foo contains the qux function. Now, if we extend the namespace Bar with a qux function then suddenly the use of qux becomes ambiguous. Worse, this ambiguity may occur in an entirely different file from where the namespace Bar is declared.

Principle 24 (No Null). *Flix does not support the null value.*

Rationale. Null — infamously dubbed the Billion Dollar Mistake by its inventor Sir Tony Hoare — is a special value that is an inhabitant of (reference) every type [25]. Today, the null value is widely considered a design mistake and languages such as C#, Dart, Kotlin and Scala are scrambling to adopt nullable type systems [16, 47, 48].

Flix, like other functional programming languages, does not have a null value, but instead relies on the Option data type to explicitly represents values that may be absent. This solution is simple to understand, works well, and guarantees the absence of the dreaded NullPointerExceptions.

Discussion. Unfortunately, this is not the entire story since Flix still needs interoperability with Java. We require that programmers explicitly check for null at the boundary between Flix and Java code. Sometimes it can be necessary to call into Java code with a null value. To support this, Flix does have a special null value of type Null, which is incompatible with any other type. Thus, to use the null value, the programmer must explicitly cast it to the desired type.

Principle 25 (No Unprincipled Overloading). *Flix does not support function overloading, i.e., the ability to define multiple functions that share the same name but have different signatures.*

Rationale. In programming languages like C++, C#, and Java, programmers can define multiple functions with the same name, as long as they take a different number of arguments and/or arguments of a different type. The relation between the overloaded functions in these languages is up to the programmer. Flix does not support such "unprincipled" overloading. Instead, Flix encourages using meaningful names for functions that share similar functionality. For example, List.join and List.joinWith, or Map.filter and Map.filterWithKey.

Flix does support *principled* overloading via type classes. For example, the Eq type class defines a signature eq (i.e., ==) which can be implemented by types that support equality. We say such overloading is *principled* because each type class lays out requirements for the overload (e.g., equality must be reflexive, symmetric, and transitive), and each type class can only be implemented once per type.

4.4 Compiler Messages

Principle 26 (The 80/20 Rule). *A compiler message must serve the programmer in two situations:*

- *as a minimally invasive hint to the programmer when he or she has made a small mistake, and*
- *as an explanatory message with lots of contextual information when the "hint" is insufficient for the programmer to understand what is wrong.*

Rationale. We call this the 80/20 rule because that 80% of the time, a software developer will have seen a specific compiler message before and require minimal information to understand and fix the underlying issue. But, 20% of the time, a software developer will encounter a compiler message he or she has not seen before and will require more information to understand and fix the problem.

For example, the message "Expected `Int32`, but found `String`" may contain enough information for the programmer to understand and fix the issue. On the other hand, the message "Unable to generalize `a -> a` to `a -> b`" may require more information for the programmer to understand what is wrong, if he or she has not seen that message before. Flix tries to achieve this with a message structure where the first sentence gets to the root of the issue and rest of the message provides more detail. Moreover, Flix has an –explain flag which can be used to obtain even more information.

Principle 27 (Compiler Message Structure). *A compiler message should have three distinct parts:*

- *Summary: A one sentence summary.*

- *Message: A multi-line text that contains all relevant details, including the program symbol(s) and fragment(s) relevant for the message.*
- *Explanation: A description of why the problem occurs and what can be done to fix it.*

Rationale. Today's compilers do not exist in isolation but are always part of a broader ecosystem. Compiler messages are not only consumed from the command line but also from integrated development environments (IDEs) and continuous integration (CI) pipelines. A compiler message must be usable in all three environments. A single-sentence summary is essential in IDEs and CIs to overview all the current messages. A full-length message is necessary to understand the details of the identified problem. Finally, a long-form and detailed explanation may be helpful for when a programmer encounters a specific error for the first time.

Principle 28 (Straight to the Point). *A compiler message should be quick to scan. Ideally, the first sentence or even the first word should contain the essence of the message.*

Rationale. A programmer must be able to quickly scan a compiler message. For example, a message like: "Duplicate definition: 'foo'" is better than the message: "The definition 'foo' is defined twice" because the programmer has to read fewer words before he or she sees the word "duplicate" which may be sufficient to understand the problem.

Discussion. Before we adopted this principle, Flix compiler messages tended to be wordy and academic. For example, a message might read "The type class 'foo' contains a duplicate definition of the signature 'bar'". While such a sentence reads nicely, it is too long. It is better to simply state the root cause: "'bar' is defined twice".

Principle 29 (Style and Tone). *A compiler message should be crisp, clear, and concise. In addition, the language should be friendly or at least neutral. Specifically, an error message should not blame the programmer.*

Rationale. A compiler should work as an assistant, not as an adversary [3, 8]. Blaming the programmer for mistakes is counter-productive. For example, the sentence "Unexpected foo" is better than "Illegal foo" because it communicates the same information without scolding the programmer. The Elm programming language goes further and tries to humanize the compiler [7]. For example, it will report, "I see Foo but was expecting to see Bar".

Discussion. Early Flix compiler messages were direct and blamed the programmer; they were a product of the way compiler-writers think: "illegal this" and "invalid that". It took a while, and the process is still ongoing, for our view to change in three significant ways:

- to ensure that compiler messages are friendly,
- to explain a problem from a programmer's point of view and not from the compiler's point of view, and
- to have the compiler retell the information it has and why it is problematic.

Principle 30 (Split Compiler Messages). *When possible, a compiler message should be broken down into several more specific compiler messages.*

Rationale. The idea is that instead of having a few generic error messages that cover all cases, we should subdivide errors into as specific messages as possible.

Discussion. Before this principle, Flix had only three type error messages: MismatchedTypes, GeneralizationError, and MissingInstance. We noticed that these three error messages can be split into more specific and detailed ones. For example,

- MismatchedTypes now has two sub-cases: OverApplied and UnderApplied for when a function is called with too many or too few arguments.
- GeneralizationError now has two sub-cases: ImpureDeclaredAsPure (an impure function is declared as pure) and EffectPolymorphicDeclaredAsPure (an effect polymorphic function is declared as pure).
- MissingInstance now has several sub-cases, including MissingEq, MissingOrder, and MissingToString. For example, the MissingToString message hints that a ToString instance can be automatically derived by adding `with ToString` to the type declaration.

Hypothesis. Giving specific type errors is more useful to programmers than giving general type errors.

Principle 31 (Relate to Other Languages). *A compiler message, when relevant, should explain how Flix relates to and differs from other programming languages.*

Rationale. Programmers learning Flix will likely be familiar with other programming languages. Therefore, it seems helpful to build on this knowledge by relating features and compiler errors to other widely used languages. For example, a compiler message can explain how the Flix type class system differs from that of Haskell or Rust. As another example, an error message related to purity or impurity can teach the programmer how the Flix effect system works.

Discussion. We think it is a missed opportunity that every programming language pretends no other languages exist! We should build on and reuse the knowledge a programmer may already have.

Hypothesis. We hypothesize that compiler messages that relate concepts, constructs, and errors to other programming languages are an effective communication form.

4.5 Standard Library

Principle 32 (Minimal Prelude). *The Flix prelude should be kept minimal and only include common functionality.*

Rationale. The Flix Prelude contains common data types and functions that are implicitly imported into the scope of every Flix file. As the Flix standard library grows, there has been a tendency for the Prelude to expand, becoming a hodgepodge of functionality. This principle aims to arrest that trend and avoid a situation like in Haskell where the Prelude is notorious for being bloated and including unsafe functions, which has led to multiple efforts to design a new and safer prelude [12, 45].

Principle 33 (Layered Abstractions). *Flix and the Flix standard library should be useable without understanding all of its features and abstractions.*

Rationale. Flix has a rich collection of functional abstractions, but a programmer should be able to gradually learn them. We envision several "levels" of programming:

- **Basic** – At the basic level, a Flix programmer writes code that uses algebraic data types, pattern matching, and recursion. A programmer may also use mutable data structures for a more imperative style of programming. In both cases, basic knowledge of the type and effect system is required[4].
- **Intermediate** – At the intermediate level, a Flix programmer writes code that relies more on the standard library. The Option, List, Set, and Map data structures are readily used. At this level, knowledge of higher-order functions and parametric polymorphism is required. Knowledge of basic type classes like Eq, Order, and ToString is also required[5].
- **Advanced** – At the advanced level, a Flix programmer can use the entire standard library, including powerful functional abstractions such as Functors, Applicatives, Monads, Monoids, and Foldables. Knowledge of higher-kinded types, type classes, and functional abstractions is required.

What is important is that Flix "duplicates" functionality to support the intermediate level. For example, Flix has both List.map and Functor.map. Since List is a Functor there is no reason to have List.map. However, by having List.map programmers do not have to know about Functor and equally important the signature of List.map is simpler and leads to better error messages. As another example, Flix has List.sum which simply computes the sum of a List[Int32]. But Flix also has List.fold which reduces a list of List[a] to a single a provided there is a Monoid instance of a. Thus, List.fold could also be used to compute the sum, but a Flix programmer is not required to know about that before they are ready.

[4]It is possible, if inelegant, to write Flix in a programming style where every function is marked as Impure. This reduces Flix to an OCaml-like or Standard ML-like language where side-effects are permitted everywhere.
[5]The Eq and Order type classes are required for Sets and Maps whereas the ToString type class is used for printing. Helpfully to the intermediate programmer, Flix supports automatic derivation of these type classes.

Principle 34 (No Dangerous Functions). *The Flix standard library should not include dangerous functions nor functions that encourage dangerous programming patterns.*

Rationale. We consider a function dangerous if it may lead to a crash. Infamously, the Haskell Prelude contains many functions that are dangerous due to their partiality, e.g. head, tail, and !!. These "functions" are not defined for all inputs and may raise exceptions at runtime, e.g. when taking the head of an empty list. Today, their inclusion in the standard library is widely considered as mistake. In Flix, our goal is that all functions in the standard library should be total. Thus functions such as head and tail return Options. As of today, the Flix standard library has 2,600 functions of which less than ten are partial.

But Flix goes further: The standard library tries to avoid functions that encourage dangerous programming patterns. For example, Flix used to have the functions: Option.isNone and Option.isSome, but such functions could mislead developers into writing code like `if (isSome(o)) /*unpack o */)` which would potentially be dangerous.

Principle 35 (Subject-Last). *Every function in the Flix standard library must accept its subject, i.e., the argument it "acts on", as its last argument.*

Rationale. The idea is to ensure a uniform standard library and to enable programmers to write pipelines. For example:

```
l |> List.map(x -> x + 123)
  |> List.filter(x -> x < 100)
  |> List.take(5)
```

The pipeline uses the `|>` operator which is really just infix function application. More importantly, the pipeline pattern works because of partial application *and* because the map, filter, and take functions are subject-last. In particular, the signature of List.map is:

```
def map(f: a -> b, l: List[a]): List[b]
```

where it is important that the list is the *last* argument.

In object-oriented programming, the this parameter is (often implicitly) passed as the first argument to a method. This principle expresses a similar sentiment but puts the "this" last in the list of arguments. Of all the Flix standard library principles, this principle has been one of the most influential and impactful since it touches on every function signature and requires us to identify its subject.

Discussion. Sometimes it can be hard to identify the subject of a function. For example, which argument to List.append is the subject? Or which argument to List.zip is the subject? Perhaps, for this reason, it is uncommon for pipelines to use these functions.

Principle 36 (Non-Commutative Functions). *A non-commutative function that accepts multiple arguments of the same type should use records to explicitly force the programmer to distinguish between the arguments.*

Rationale. When a function takes two arguments of the same type there is potential for confusion: which argument is which? For example, in the call `String.contains(s1, s2)` which argument is the haystack and which is the needle? Mistakenly swapping the two arguments looks innocuous and type checks but is incorrect. Many programming languages attempt to overcome the confusion with labelled arguments. But labelled arguments, in their typical implementation, has at least two downsides:

- they are optional; hence programmers may forget to use them, and
- they are typically not part of the type system, hence they do not work with first-class functions.

Flix overcomes both problems by using records to implement labelled arguments. If a function requires a record argument, it becomes part of its type, and every call site must pass a record, making it explicit which arguments are which.

Discussion. We could give String.contains the signature:

```
def contains(args: {needle = String,
                    haystack = String}): Bool =
```

which can be called as:

```
contains({needle = "a", haystack = "abc"})
```

Unfortunately, this signature does not work with pipelines:

```
"Hello" :: "Bonjour" :: "Guten Tag" :: Nil
    |> List.map(String.toLowerCase)
    |> List.map(String.contains(needle = "Hola"))
    |> println
```

because the call to String.contains expects a record with two fields: needle and haystack and cannot be partially applied to a record with only one field. This situation is unacceptable because it breaks support for pipelines. However, perhaps surprisingly, there is a good solution: We require every argument *except the last* (i.e., the subject) to be a record. The updated signature of `String.contains` is:

```
def contains(needle: { needle = String },
             haystack: String): Bool =
```

The contains function now takes a singleton record as its first argument and a regular string as its second argument. The two arguments cannot be confused because a record and a string are not the same type. We call the new contains as:

```
contains({needle = "a"}, "abc")
```

which, using syntactic sugar, can be shortened to:

```
contains(needle = "a", "abc")
```

We have achieved three objectives:

- We cannot mistakenly swap the arguments to contains because one of them must be a record.
- At every call site, we can immediately identify which string is the needle and which is the haystack because the former is always explicitly named.
- We have retained the ability to use pipelines.

We have applied this principle to the Flix standard library. In total 66 functions out of 2,600 functions use this principle. Note that the principle only applies to non-commutative functions, e.g. it does not apply to `Set.union` because the order of its arguments is immaterial.

Limitations. An important question that arose during the refactoring of the library was whether to apply this principle indiscriminately. For example, Flix has a function `List.append`, which is not commutative. Nevertheless, it seems intuitive that the "left" argument is appended to the "right" argument. Similarly, for `List.zip`. Consequently, we decided to apply the principle on a case-by-case basis to the standard library.

Principle 37 (Symbolic Operators Must Have Names). *Flix supports symbolic operators, i.e., defining functions with names such as `+`, `++`, `|+|`, and so forth, but requires that every symbolic function should also be given a "pronounceable" ASCII name, e.g. "plus", "append", and so on.*

Rationale. Flix tries to strike a balance between conciseness and understandability. Expert programmers can define and use symbolic operators when it makes their source code compact and easy to understand (from an expert point-of-view). However, at the same time, they must also provide human-readable names accessible to non-expert programmers.

To give two examples, the List.append function can be referred to as `:::`, and the SemiGroup.combine function can be referred to as `++`.

Principle 38 (Destructive Operations are Marked). *In Flix, every destructive operation is suffixed with "!".*

Rationale. Flix supports both destructive and non-destructive operations on mutable data structures. For example, the `Array.reverse(a)` function returns a copy of the array a with the elements in reverse order, whereas `Array.reverse!(a)` destructively re-orders the elements of a. The purpose of this principle is two-fold:

- to ensure that Flix programmers can readily determine whether they are using a destructive or non-destructive operation on mutable data, and
- to provide a consistent naming scheme for operations that exists in both destructive and non-destructive flavours (e.g. reverse vs. reverse!).

Discussion. This principle, which is inspired by Scheme, applies to destructive and non-destructive operations on *mutable data*. In particular, it *does not* apply to all impure functions nor to functions that modify the outside world. For example, println and deleteFile do not end in an exclamation mark, despite their impure behavior. Currently the Flix type and effect system tracks reads and writes to mutable memory, but it does not distinguish between them. There is on-going discussion on whether to change that. If so, this principle might become enforceable by the compiler.

Principle 39 (Mirrored Names of Destructive and non-Destructive Operations). *In Flix, non-destructive and destructive operations that share similar behavior and similar type signatures should share similar names.*

Rationale. This principle aims to ensure that similarly named operations have similar type signatures. For example, Array.reverse and Array.reverse! share similar signatures, except one returns a new array and the other destructively updates the array. On the other hand, while there is a Array.map there is no Array.map! because the signature of Array.map is: `(a -> b) -> Array[a] -> Array[b]` but the signature of the destructive update cannot take a function from `a -> b` because the type of an array cannot change once constructed. Instead, Flix offers a Array.transform! operation with the signature: `(a -> a) -> Array[a] -> Unit`.

4.6 Miscellaneous Principles

Principle 40 (Annotation vs. Modifiers). *Flix has* annotations *and* modifiers. *We define an* annotation *as a specific piece of meta-data associated with a specific construct (e.g., a function or type declaration). We define a* modifier *as a keyword that affects the semantics of a program.*

Rationale. In Flix, a modifier may impact both the semantics and well-formedness of a program. For example, the pub access modifier controls name resolution; marking a construct as private (i.e., non-pub) may cause a program to no longer compile. As another example, the override modifier is *required* to make a type-class instance well-formed when it overrides a function from the class. In Flix, annotations, on the other hand, have no impact on semantics or well-formedness. Instead, they are purely a mechanism to attach meta-data to a programming construct.

Principle 41 (Annotations are Built-in). *In Flix, annotations are built-in; they cannot be user defined.*

Rationale. Annotations typically have two use cases: (i) as an ad-hoc mechanism to document program constructs (e.g., classes, methods), and (ii) as a mechanism to support reflective programming. Flix does not have reflection and does not need annotations to support that use case. Flix does support a small collection of built-in annotations for documentation.

The issue with user-defined annotations is that it is hard to ensure they have well-defined semantics and are only declared once. For example, Java has @Deprecated which is defined in the Java standard library, has a well-defined meaning, and is widely used. On the other hand, Java did not define a @NonNull annotation with the consequence that there are at least eight different variants in use, and each has subtly different semantics. We want to avoid such a situation by making annotations part of the language. If new annotations are needed, they can be added after discussion.

5 Abandoned Principles

We conclude with a discussion of principles we had adopted but that we later had to abandon. We do not believe these principles are broken, but they did not work out for Flix.

Aban. Principle (No Blessed Library). *The Flix programming language and its compiler should be independent of the Flix standard library.*

Rationale. The idea behind this principle was to ensure a clear separation between the Flix language and compiler, and the Flix standard library. In particular, a programmer should be able to write a program that does not depend on the standard library and also to substitute his or her own standard library, if so desired.

Discussion. Initially, while the standard library was small, the separation between language and library was straightforward to enforce. But, as the language grew, the separation became more and more challenging to enforce.

We will give a few examples. The type classes Eq, Order, and ToString are declared in the library, but the compiler needs to know about them for several reasons: (i) to support translation of == into a call to Eq.eq, (ii) to support translation of string interpolators into calls to ToString, (iii) to support automatic derivation of Eq, Order, and ToString, and (iv) for the compiler to give more specific error messages related to missing instances of these classes. As similar set of dependencies arose between the logic part of the language and dependencies on the type classes that define a lattice.

Over time it simply became "too convenient" for the compiler to know about the standard library; the upside is greatly improved usability, but the downside is a hard dependence between language and compiler.

Aban. Principle (Uniform Function Call Syntax). *In Flix, the syntax of a function call is* `f(a, b, c)`. *With UFCS, the same function call can be written with syntactic sugar and in an object-oriented style as:* `a.f(b, c)`.

Rationale. The idea behind uniform function call syntax (UFCS) [10, 55] is three-fold:

- to emulate the syntax of a method call, i.e., `o.f()`,
- to enable "auto-completion on the dot", and
- to enable method chaining, i.e., `x.f().g()`.

The latter is sometimes referred to as "fluent-style" [18] or inaccurately as the "builder-pattern". All three reasons seem useful and we speculate that they may enable object-oriented programmers to feel more at home in a functional programming language.

Discussion. UFCS is good in that it does not confuse syntax with semantics (Principle 1). D and Nim support UFCS, and there is discussion of adding it to C++. Flix used to support UFCS, but we had to abandon it for two reasons:

(a) It is ambiguous in the presence of records.
(b) It requires the subject to be the first argument.

For (a), the problem is that the expression `a.f(b, c)` can be parsed as both: (i) `(a.f)(b, c)` (i.e., a field access of f which is then invoked), and (ii) as the UFCS function call `f(a, b, c)`. Which is right? Which would the programmer expect? For (b), the problem is that a call `a.f(b, c)` is only sensible if a is the subject. But, if we have e.g. `List.map(f: a -> b, l: List[a])`, where the function argument is first, then we cannot write `l.map(x -> x + 1)` as we wanted. Thus we have to change the signature to `List.map(l: List[a], f: a -> b)`, but this now prevents partial application like `List.map(x -> x + 1)`.

The fundamental problem is a tension between subject-first or subject-last. We cannot have both. We briefly considered whether to treat a UFCS call `a.f(b, c)` as `f(c, b, a)`, but this was quickly abandoned, and ultimately we dropped UFCS support and settled on the principle of "subject-last" to fully support pipelines.

6 Related Work

We conclude the paper by presenting related work that has had a significant impact on the design of Flix.

Human-Computer Interaction. The inspiration for Flix' principles on compiler error messages came partly from two the influential blog posts: "Compiler Errors for Humans" [7] and "Compilers as Assistants" [8] written by Evan Czaplicki, the lead developer of the Elm programming language [17]. In these two posts, Evan lays out his vision for "compilers as assistants, not as adversaries". We were also inspired by a recent study by [3] which discusses the use of a positive tone, among many other aspects, in the design of error messages.

Type System. The Flix type and effect system is based on Hindley-Damas-Milner, the same type system that powers Standard ML, OCaml, and Haskell [9, 22, 46, 60]. As discussed, this system supports let-polymorphism and complete type inference with Algorithm W [43]. Nevertheless, Flix has made the design choice to require all top-level function to be annotated with type signatures. As stated in Principle 7, this has three advantages: type signatures are useful as documentation, to accurately assign blame, and they enable parallel type inference.

Type Classes. Flix supports higher-kinded types and type classes inspired by Haskell [58]. The current implementation is close to that of Haskell 98 [29]. Every type class system comes with a range of design choices and the paper "Type classes: an exploration of the design space" [30] was instrumental in guiding our design. Even more fundamental was the "Typing Haskell in Haskell" [28] paper which we have used as the blueprint for the type inference algorithm in Flix. This is probably the single most important paper that has influenced the implementation of Flix. We wish there were more papers like it; papers that go into great detail to describe the implementation of complex compilers.

Effect System. Flix has a type and effect system based on Boolean unification [4, 6, 38, 44]. The effect system supports effect polymorphism [34], i.e., the effect(s) of a higher-order function may depend on the effect(s) of its function arguments. Unlike algebraic effect systems [32, 33], which allows user-defined effects, the Flix effect system is focused on precise tracking of purity. In particular, as shown in Section 2, the effect system supports a form of region-based memory management [57]. In Flix regions are not used for static garbage collection nor are they automatically inferred [56], instead they are used by the programmer to delimit use of mutable memory allowing pure functions to use an imperative-style internally.

Retrospectives. Several of our principles were inspired by the reflections of other programming language designers on mistakes or design flaws in their languages. Famously, Tony Hoare gave a talk on "Null references: The billion dollar mistake" [25]. Graydon Hoare, the inventor of Rust, wrote an influential blog post entitled "Things Rust Shipped Without" [24] where he tried to enumerate all the design mistakes that he believed Rust 1.0 had avoided (including null pointers). More recently, Troels Henriksen, the designer of Futhark [20, 21], wrote a blog post "Design Flaws in Futhark" [19] which was widely discussed on the /r/programminglanguages sub-Reddit and inspired several similar posts by other language designers. We have used many of these posts and discussions as a resource for design mistakes that we do not want to repeat in Flix.

The Flix Programming Language. Flix itself has been the subject of multiple research papers [35–41]. A line of research has focused on the Datalog aspect of Flix, including its support for first-class Datalog constraints and lattice semantics [37, 40], and how to use these features to declaratively express sound program analyses in Flix [36, 41]. Another line of research has focused on the use of Boolean unification in type inference with applications in effect systems [38] and for relational nullable types [39]. Flix has also been covered by the tech media, with articles in InfoQ [27], Version2 [1], and ComputerWorld [54].

7 Conclusion

We have presented the *values* and *principles* that underpin the design of the Flix programming language. We have described the rationale for each principle and discussed how it has shaped the development of Flix.

We that hope these principles will inspire programming language designers and stimulate more discussion on the 'softer' aspects of programming language design.

Acknowledgments

The author would like to thank Matthew Lutze, Jonathan L. Starup, and Jaco van de Pol for many useful discussions.

References

[1] Tania Andersen. 2021. *Flix: Nyt sprog fra Aarhus vil gøre programmørens liv lettere med logik i tanken.* https://www.version2.dk/artikel/flix-nyt-sprog-fra-aarhus-vil-goere-programmoerens-liv-lettere-med-logik-i-tanken

[2] Andrew Gerrand. 2022. *Share Memory By Communicating.* https://go.dev/blog/codelab-share

[3] Brett A Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael Osera, et al. 2019. Compiler Error Messages Considered Unhelpful: The Landscape of Text-Based Programming Error Message Research. In *Proc. of the Working Group Reports on Innovation and Technology in Computer Science Education.* https://doi.org/10.1145/3344429.3372508

[4] George Boole. 1847. *The Mathematical Analysis of Logic.*

[5] Gert-Jan Bottu, Ningning Xie, Koar Marntirosian, and Tom Schrijvers. 2019. Coherence of Type Class Resolution. *Proc. of the ACM on Programming Languages* 3, ICFP (2019). https://doi.org/10.1145/3341695

[6] Alexandre Boudet, Jean-Pierre Jouannaud, and Manfred Schmidt-Schauß. 1989. Unification in Boolean Rings and Abelian Groups. *Journal of Symbolic Computation* (1989). https://doi.org/10.1016/S0747-7171(89)80054-9

[7] Evan Czaplicki. 2015. *Compiler Errors for Humans.* https://elm-lang.org/news/compiler-errors-for-humans

[8] Evan Czaplicki. 2015. *Compilers as Assistants.* https://elm-lang.org/news/compilers-as-assistants

[9] Luis Damas. 1984. *Type Assignment in Programming Languages.* Ph. D. Dissertation. The University of Edinburgh.

[10] D Language Developers. 2022. *D Language Specification.* https://dlang.org/spec/function.html#pseudo-member

[11] The Pony Developers. 2021. *Divide by Zero.* https://tutorial.ponylang.io/gotchas/divide-by-zero.html

[12] Stephen Diehl. 2016. *relude: Safe, performant, user-friendly and lightweight Haskell Standard Library.* https://hackage.haskell.org/package/relude

[13] Edsger Wybe Dijkstra et al. 1970. Notes on Structured Programming.

[14] Joe Duffy. 2018. *The Error Model.* http://joeduffyblog.com/2016/02/07/the-error-model/

[15] Jana Dunfield and Neel Krishnaswami. 2021. Bidirectional Typing. *ACM Computing Surveys (CSUR)* (2021). https://doi.org/10.1145/3450952

[16] Manuel Fähndrich and K Rustan M Leino. 2003. Declaring and Checking Non-Null Types in an Object-Oriented Language. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).* https://doi.org/10.1145/949305.949332

[17] Richard Feldman. 2020. *Elm in Action.* Simon and Schuster.

[18] Martin Fowler. 2005. *FluentInterface.* https://martinfowler.com/bliki/FluentInterface.html

[19] Troels Henriksen. 2019. *Design Flaws in Futhark.* https://futhark-lang.org/blog/2019-12-18-design-flaws-in-futhark.html

[20] Troels Henriksen, Niels GW Serup, Martin Elsman, Fritz Henglein, and Cosmin E Oancea. 2017. Futhark: Purely Functional GPU-Programming with Nested Parallelism and In-Place Array Updates. In *Programming Language Design and Implementation (PLDI).* https://doi.org/10.1145/3062341.3062354

[21] Troels Henriksen, Frederik Thorøe, Martin Elsman, and Cosmin Oancea. 2019. Incremental Flattening for Nested Data Parallelism. In *Principles and Practice of Parallel Programming (PPoPP).* https://doi.org/10.1145/3293883.3295707

[22] Roger Hindley. 1969. The Principal Type-Scheme of an Object in Combinatory Logic. *Transactions of the American Mathematical Society (AMS)* (1969).

[23] Charles Antony Richard Hoare. 1978. Communicating Sequential Processes. *Commun. ACM* (1978).

[24] Graydon Hoare. 2015. *Things Rust Shipped Without.* https://graydon2.dreamwidth.org/218040.html

[25] Tony Hoare. 2009. Null References: The Billion Dollar Mistake. *Presentation at QCon London* (2009).

[26] David Hovemeyer and William Pugh. 2004. Finding Bugs is Easy. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).* https://doi.org/10.1145/1052883.1052895

[27] Johan Janssen. 2022. *Interview with Magnus Madsen about the Flix Programming Language.* https://www.infoq.com/news/2022/02/flix-programming-language/

[28] Mark P Jones. 1999. Typing Haskell in Haskell. In *Haskell Workshop.*

[29] Simon Peyton Jones. 2003. *Haskell 98 Language and Libraries: The Revised Report.* Cambridge University Press.

[30] Simon Peyton Jones, Mark Jones, and Erik Meijer. 1997. Type classes: an exploration of the design space. In *Haskell Workshop.*

[31] Daan Leijen. 2005. Extensible Records with Scoped Labels. *Trends in Functional Programming* (2005).

[32] Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types. *arXiv* (2014).

[33] Daan Leijen. 2017. Type Directed Compilation of Row-Typed Algebraic Effects. In *Principles of Programming Languages (POPL).* https://doi.org/10.1145/3093333.3009872

[34] John M Lucassen and David K Gifford. 1988. Polymorphic Effect Systems. In *Principles of Programming Languages (POPL).*

[35] Magnus Madsen and Ondřej Lhoták. 2018. Implicit Parameters for Logic Programming. In *Principles and Practice of Declarative Programming (PPDP).* https://doi.org/10.1145/3236950.3236953

[36] Magnus Madsen and Ondřej Lhoták. 2018. Safe and Sound Program Analysis with Flix. In *International Symposium on Software Testing and Analysis (ISSTA).* https://doi.org/10.1145/3213846.3213847

[37] Magnus Madsen and Ondřej Lhoták. 2020. Fixpoints for the Masses: Programming with First-Class Datalog Constraints. *Proc. of the ACM on Programming Languages* 4, OOPSLA (2020). https://doi.org/10.1145/3428193

[38] Magnus Madsen and Jaco van de Pol. 2020. Polymorphic Types and Effects with Boolean Unification. *Proc. of the ACM on Programming Languages* 4, OOPSLA (2020). https://doi.org/10.1145/3428222

[39] Magnus Madsen and Jaco van de Pol. 2021. Relational Nullable Types with Boolean Unification. *Proc. of the ACM on Programming Languages* 5, OOPSLA (2021). https://doi.org/10.1145/3485487

[40] Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. 2016. From Datalog to Flix: A Declarative Language for Fixed Points on Lattices. In *Programming Language Design and Implementation (PLDI).* https://doi.org/10.1145/2980983.2908096

[41] Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. 2016. Programming a Dataflow Analysis in Flix. In *Tools for Automatic Program Analysis (TAPAS).*

[42] Magnus Madsen, Ramin Zarifi, and Ondřej Lhoták. 2018. Tail Call Elimination and Data Representation for Functional Languages on the Java Virtual Machine. In *Compiler Construction (CC).* https://doi.org/10.1145/3178372.3179499

[43] Alberto Martelli and Ugo Montanari. 1982. An Efficient Unification Algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)* (1982).

[44] Urusula Martin and Tobias Nipkow. 1989. Boolean Unification - The Story So Far. *Journal of Symbolic Computation* (1989).

[45] Daniel Mendler. 2022. *intro: Safe and minimal prelude.* https://hackage.haskell.org/package/intro

[46] Robin Milner. 1978. A Theory of Type Polymorphism in Programming. *J. Comput. System Sci.* (1978).

[47] Abel Nieto, Marianna Rapoport, Gregor Richards, and Ondřej Lhoták. 2020. Blame for Null. In *European Conference on Object-Oriented Programming (ECOOP 2020).* https://doi.org/10.4230/LIPIcs.ECOOP.2020.3

[48] Abel Nieto, Yaoyu Zhao, Ondřej Lhoták, Angela Chang, and Justin Pu. 2020. Scala with Explicit Nulls. In *European Conference on Object-Oriented Programming (ECOOP 2020)*. https://doi.org/10.4230/LIPIcs.ECOOP.2020.25

[49] Martin Odersky. 2018. *Opening Keynote: Preparing for Scala 3 by Martin Odersky and Adriaan Moors*. https://www.youtube.com/watch?v=1VDOhiFYW3Y

[50] Benjamin C Pierce and David N Turner. 2000. Local Type Inference. *ACM Transactions on Programming Languages and Systems (TOPLAS)* (2000). https://doi.org/10.1145/345099.345100

[51] Rich Hickey. 2022. *Simple Made Easy*. https://www.infoq.com/presentations/Simple-Made-Easy

[52] Scala. 2022. *Scala Documentation — Implicit Classes*. https://docs.scala-lang.org/overviews/core/implicit-classes.html

[53] Michel Schinz and Martin Odersky. 2001. Tail Call Elimination on the Java Virtual Machine. *Electronic Notes in Theoretical Computer Science* (2001). https://doi.org/10.1016/S1571-0661(05)80459-1

[54] Jakob Schjoldager. 2021. *Datalogi-adjunkt Magnus Madsen har opfundet et nyt programmeringssprog: Vi står over for et skifte inden for programmeringssprog - her er ideen med det nye Flix*. https://www.computerworld.dk/art/257120/datalogi-adjunkt-magnus-madsen-har-opfundet-et-nyt-programmeringssprog-vi-staar-over-for-et-skifte-inden-for-programmeringssprog-her-er-ideen-med-det-nye-flix

[55] Herb Sutter. 2014. *Unified Call Syntax*. https://isocpp.org/files/papers/N4165.pdf

[56] Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. 2004. A Retrospective on Region-based Memory Management. *Higher-Order and Symbolic Computation* (2004).

[57] Mads Tofte and Jean-Pierre Talpin. 1997. Region-based Memory Management. *Information and Computation* (1997).

[58] Philip Wadler and Stephen Blott. 1989. How to make ad-hoc polymorphism less ad hoc. In *Principles of Programming Languages (POPL)*. https://doi.org/10.1145/75277.75283

[59] Wikipedia. 2022. *Extension Method — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/Extension_method

[60] Andrew K Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Information and Computation* (1994). https://doi.org/10.1006/inco.1994.1093

[61] Yichen Xie and Dawson Engler. 2002. Using Redundancies to Find Errors. In *Foundations of Software Engineering (FSE)*. https://doi.org/10.1145/587051.587060

Received 2022-07-12; accepted 2022-10-02