Four Fascinating Programming Languages

(You've Probably Never Heard Of)

Lutz Hühnken — BOBKonf 2026

Sharing welcome!

If you post about this talk, please tag me:

@huehnken.de

@lutzhuehnken@mastodon.social

#bobkonf

Four Concepts

1. Algebraic Effects: composable effect handlers with purity reflection

2. Content-Addressed Code: definitions identified by hash, not by name or file

3. Concurrent Logic Programming: computation as communicating processes synchronized through dataflow variables

4. Failable Expressions: failure as control flow with speculative execution and rollback

The Languages

Flix

Algebraic Effects

🎓 Academic

Unison

Content-Addressed Code

🤝 Non-profit

Strand

Concurrent Logic Programming

📜 Historic

Verse

Failable Expressions

(and more)

🏢 Commercial

Flix

“What if the effect system could reason about purity and use it to optimize automatically?”

Created by **Magnus Madsen** at Aarhus University

Compiles to **JVM bytecode** with full Java interop

Inspired by **OCaml, Haskell, Rust** and **Scala**

flix.dev

The Effects Problem

Every real program does more than compute: it reads files, prints output, throws errors, manages state. These are **effects**.

Languages differ in **whether and how they track effects** in the type system.

Most languages

Don't track effects at all. Any function can do anything. You just have to know.

Some languages

Track effects in the type system, but the mechanisms vary considerably.

Haskell: Monads

Haskell separates pure from effectful code using the **IO monad**:

```haskell
greet :: String -> IO ()          -- type says: this does I/O
greet name = putStrLn ("Hello, " ++ name)

add :: Int -> Int -> Int          -- no IO: guaranteed pure
add x y = x + y
```

This enforces a clear separation. But monads do not compose: combining IO with other effects (failure, state, readers) requires **monad transformers**.

Effects as Sets

Flix tracks effects in every type signature. The syntax uses `\` after the return type:

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

def add(x: Int32, y: Int32): Int32 =    // no annotation: pure
    x + y

def parseAndLog(s: String): Int32 \ {IO, Parse} =  // multiple effects
    let n = parse(s);
    println("Parsed: ${n}");
    n
```

Effects form a **set** with algebraic operations — union, intersection, complement, and difference.

Effect Polymorphism and Exclusion

Effect variables propagate through higher-order functions:

```
def map(f: a → b \ ef, l: List[a]): List[b] \ ef = ...

map(x → x + 1, myList)                // inferred: pure
map(x → {println(x); x + 1}, myList)  // inferred: IO
```

Set algebra enables **effect exclusion** — constraining which effects are not allowed:

```
// The handler h may have any effect EXCEPT Throw
def recoverWith(f: Unit → a \ Throw,
                h: ErrMsg → a \ (ef - Throw)): a \ ef = ...
```

The type system guarantees that the recovery handler **cannot itself throw**.

Purity Reflection

A higher-order function can **inspect at runtime** whether its function argument is pure, and choose a different strategy:

```
def count(f: a → Bool \ ef, s: Set[a]): Int32 \ ef =
    match purityOf(f) {
        case Purity.Pure(g)   ⇒
            // safe to parallelize — no side effects
            Set.parCount(g, s)
        case Purity.Impure(g) ⇒
            // must stay sequential — has side effects
            foldLeft((b, k) → if (g(k)) b + 1 else b, 0, s)
    }
```

When callers pass a pure function, the implementation **automatically uses parallel evaluation**. Effectful functions fall back to sequential.

Algebraic Effects

Declare effects with **operations** (like an interface) Example

Functions declare which effects they use — effects **compose freely**

Handlers provide the interpretation and can **resume** Example

Think **“resumable exceptions”**: when you “throw” an effect, the handler can send a value back and continue

Effects are declared where they are used and handled wherever appropriate.

Effects Across Languages

Flix isn't alone. The idea of tracking effects is spreading:

Unison — Abilities

Same concept, different name. Unison calls them "abilities" and uses them for everything from I/O to distributed computing.

Verse — <decides> <transacts>

Effect annotations mark failable and transactional code. The runtime uses them for automatic rollback.

OCaml 5

Added algebraic effect handlers in 2022. The first widely-used language to adopt them.

Koka

A pioneering effect system by Daan Leijen (MSR). Uses row-based effects rather than Flix's set-based approach.

Also Worth Knowing

Region-Based Local Mutation

Pure functions can use mutable arrays and references internally via scoped regions. Mutation cannot escape the region. Example

First-Class Datalog

Datalog constraints are embedded directly in the language for relational queries and fixed-point computations. Example

Type System

Type Classes (called Traits), Higher-Kinded Types, Hindley-Milner inference for both types and effects.

CSP Style Concurrency

Channels, Coroutines (called spawned processes)

Unison

“What if code were identified by content, not by name or file path?”

Created by **Paul Chiusano**, **Runar Bjarnason** & **Arya Irani**
at Unison Computing

Statically-typed, purely functional
Inspired by **Haskell** and **Erlang**

unison-lang.org

Content-Addressable Code

In every other language, code lives in **text files** and is identified by **name and path**

In Unison, every definition is **hashed** by its AST. The hash is its identity.

Names are just **metadata** — human-readable labels pointing to hashes

Code is stored in a **database** (the codebase), not in text files

Renaming is free

Change the label, the hash stays the same. All references remain valid.

No build step

Code is stored already type-checked. There is no separate compilation phase.

No dependency conflicts

Dependencies are pinned by hash, not version string. Different versions coexist naturally.

Tests are cached

Pure tests only re-run when their dependencies actually change.

How It Feels

```
-- scratch.u (you write code in a scratch file)
square : Nat → Nat
square x = x * x

-- The UCM picks it up automatically:
-- ⍟ These new definitions are ok to `add`:
--     square : Nat → Nat

-- Under the hood, square gets hash #a5f2bc ...
-- Now rename it:
.> move.term square squareOf

-- Nothing breaks. All dependents still reference #a5f2bc ...
-- The name just changed.
```

There's no "find and replace" across files. The hash never changed, so **nothing needs updating**.

Distributed Computing as a Consequence

When code is identified by hash, you can **send a hash to a remote node** and it fetches the definition on demand

“Distributed programming as a library” — not a separate infrastructure layer

```
-- Conceptual Unison
Remote.at node2 '(expensive-computation data)
-- The runtime ships the function hash to node2
-- node2 fetches the definition, runs it, returns result
```

A single Unison program can describe an **entire distributed system**.

Also Worth Knowing

Abilities (Effect System)

Unison's version of algebraic effects. Same idea as Flix: declare, use, handle — with the same composability benefits.

Unison Cloud

A platform for deploying distributed Unison programs. The language and the infra are one.

Structured Version Control

Merge conflicts are semantic, not textual. No conflicts from whitespace or import order.

Strand

“What if programs were networks of concurrent processes, communicating through logical variables?”

Created by **Ian Foster** & **Stephen Taylor**

Prototype 1986 at **Imperial College London**
Book published 1990

A **concurrent logic** language
BCS Award for Technical Innovation 1989

Foster & Taylor, “Strand: New Concepts in Parallel Programming” (Prentice Hall, 1990)

Concurrent Logic Programming

Conventional programs execute as a **single thread** of sequential instructions

A Strand program is a **set of concurrent processes**, each defined by guarded clauses

Processes communicate through **single-assignment variables**: a variable can be bound at most once

Reading an unbound variable **suspends** the reading process until a value is provided — dataflow synchronization

Concurrency is **inherent in the programming model**, not layered on top with threads and locks.

Processes Communicating Through Variables

⇄ Concurrent

```
do_both(R1, R2) :-
    compute_a(R1),
    compute_b(R2).
```

☰ Sequential

```
do_sequence(R) :-
    compute_a(X),
    compute_b(X, R).
```

compute_a and compute_b run as **concurrent processes**. The shared variable synchronizes them automatically.

Portable Parallelism

The same Strand program runs on fundamentally different parallel architectures:

Architecture	Example
Transputer networks	INMOS computing surfaces
Hypercubes	Intel iPSC
Shared memory	Sequent Balance/Symmetry
Workstation clusters	Sun networks via LAN

The runtime maps processes to processors. The programmer writes **architecture-independent concurrent logic**.

Also Worth Knowing

Prolog syntax, different semantics

Looks like Prolog, but instead of predicates holding it's about processes terminating.

Influenced later concurrent systems

Ideas from Strand and concurrent logic programming fed into Erlang, and dataflow architectures.

Real applications in the late 1980s

Weather modeling, protein structure prediction, parallel theorem proving — on actual parallel hardware.

"Extinct"

No modern implementation. To experiment: Strand in Forth

Ian Foster later co-invented the computational grid and co-authored the Globus toolkit.

Verse

“What if failure wasn’t an exception to handle, but a natural part of how expressions evaluate?”

Designed by **Tim Sweeney** (founder/CEO of Epic Games)

Formalized by **Simon Peyton Jones** & **Lennart Augustsson**

A **functional logic** language inside the Fortnite ecosystem

dev.epicgames.com/documentation/en-us/fortnite/verse-language-reference

Failure as Control Flow

In most languages, `if` checks a boolean. Exceptions are thrown and caught. **Separate mechanisms.**

In Verse, everything is an expression that either **succeeds** (produces a value) or **fails** (produces nothing)

No booleans for control flow. `x < 10` is **failable** — it succeeds if true, fails if false

Failable expressions run inside **failure contexts** (like `if` blocks)

When something fails, all side effects are **rolled back automatically** — speculative execution

Failure in Action

```
# Failable array access: no index-out-of-bounds errors!
if (Element := MyArray[Index]):
    Log(Element)           # Only runs if Index was valid
else:
    Log("Index invalid")   # Failure path

# Chained failable operations with automatic rollback
var Gold : int = 100
var Inventory : []item = array{}

PurchaseItem(Item : item, Cost : int)<transacts><decides> : void =
    set Gold -= Cost           # Tentatively deduct gold
    Gold ≥ 0                   # Failable: fails if gold went negative
    set Inventory += array{Item}  # Add item

# If Gold ≥ 0 fails, the Gold deduction is rolled back!
```

Failure propagates naturally through expressions, and **mutations within failure contexts are rolled back automatically**.

The <decides> Effect

```
# The <decides> effect marks functions that can fail
GetRarityMultiplier(Rarity : string)<decides> : float =
    if (Rarity = "Common"):    return 1.0
    if (Rarity = "Rare"):      return 1.5
    if (Rarity = "Legendary"): return 3.0
    false?   # Explicit failure: unknown rarity

# The <transacts> effect enables automatic rollback
Transfer(From : account, To : account, Amount : int)<transacts><decides> : void =
    set From.Balance -= Amount
    From.Balance ≥ 0    # Fails (and rolls back) if insufficient
    set To.Balance += Amount
```

The effect system tells you at a glance: this function **might fail**, and this function does **speculative mutation**.

Effect Exclusion

`<no_rollback>` marks operations with **irreversible** side effects:

```
SendEmail(To : string, Body : string)<no_rollback> : void = ...
LaunchMissile()<no_rollback> : void = ...
```

The compiler **rejects** `<no_rollback>` inside a `<transacts>` context:

```
Transfer(From : account, To : account, Amount : int)<transacts><decides> : void =
    set From.Balance -= Amount
    SendEmail(To.Owner, "Transfer received!")  # COMPILE ERROR!
    From.Balance ≥ 0                           # might fail and roll back
    set To.Balance += Amount                   #  ...but the email is already sent
```

Same idea as Flix's effect exclusion — the type system **statically prevents mixing incompatible effects**.

Lenient Evaluation

```
x := SlowOk()     # takes 10 units, succeeds
y := FastFail()   # takes 1 unit, FAILS
z := Use(x, y)    # needs both x and y
```

Strict

```
t=0  start SlowOk
t=10 SlowOk done
     start FastFail
t=11 FastFail fails
```

Wasted: **10 units**

Lazy

```
t=0  z demands x
t=10 SlowOk done
     z demands y
t=11 FastFail fails
```

Wasted: **10 units**

Lenient

```
t=0  start both
t=1  FastFail fails
     abandon SlowOk
```

Wasted: **1 unit**

Lazy pulls from one end. Lenient **pushes from all unblocked fronts simultaneously** — and can abandon expensive work early.

Types as Failable Functions

In Verse, a type is a failable function. Applying `int` to a value **succeeds** if it's an integer, **fails** otherwise.

```
# Type checking is function application:
if (N := int[SomeValue]):
    DoSomethingWith(N)     # N is known to be an int

# So you can define your own types the same way:
PositiveInt(X : int)<decides> : int =
    X > 0    # failable
    X

EvenInt(X : int)<decides> : int =
    Mod[X, 2] = 0
    X
```

Type checking, pattern matching, and validation are **the same operation**: apply a failable function.

`for` as Quantifier

Counting adjacent mines in a grid — the entire nested iteration, filtering, and counting is a single `for` expression:

```
for:
    Y→CellRow : Cells
    X→Cell : CellRow
    AdjacentX := X-1..X+1
    AdjacentY := Y-1..Y+1
    AdjacentCell := Cells[AdjacentY][AdjacentX]
    Cell <> AdjacentCell
    AdjacentCell.Mined?
do:
    set Cell.AdjacentMines += 1
```

Each line is a **constraint**. Failing lines (out-of-bounds access, self-comparison, non-mined cell) are silently skipped — not errors.

This is **quantification over a solution space**, not nested loops with bounds checking.

Also Worth Knowing

`for` as Logical Quantifier

`for (X : Candidates, X > 5)` is not iteration — it's quantification over a solution space. The output is the set of all solutions.

`one{}` and `all{}`

Verse code can be nondeterministic. `one{}` commits to the first solution (like Prolog's cut). `all{}` collects every solution. You choose how much of the search space to resolve.

Types as Values

Types can be manipulated at runtime, not just compile time.

Gradual Typing

Mix static and dynamic typing, with refinement types for stronger guarantees.

Time-Dependent Execution

Expressions can span multiple simulation frames. Combinators like `Race`, `Rush`, `Sync`, and `Branch` compose temporal behavior.

Failable expressions. Lenient evaluation. Logic variables.
Speculative execution with automatic rollback. Types as functions.

Arguably the most ambitious language design of the decade.

Formalized by Simon Peyton Jones and Lennart Augustsson.
The team includes multiple CS professors and PL researchers.

It ships inside the Fortnite SDK.

What We Learned

Concept	Language	Approach
Side effects are primitives	`Flix`	Set-based effect algebra with purity reflection
Code = names + files	`Unison`	Content-addressable, hash-identified code
Programs = sequential threads	`Strand`	Concurrent logic programming with dataflow synchronization
Execution goes forward	`Verse`	Failable expressions with speculative rollback

Thank You

To explore further:

Flix	Unison	Strand	Verse
flix.dev	unison-lang.org		dev.epicgames.com

https://programminglanguages.eu