Modern Language Design

Fabio Pellacini, FIM, UniMoRe

Language Design Goals have not changed

Efficiency

Reliability

Productivity

		Language	Avg	StDev
1		JavaScript	1.75	0.95
2		Python	1.75	0.95
3		TypeScript	4	1.82
4		Java	4.5	1.73
5		C#	6	1.41
6		C++	6.75	1.7
7		PHP	7.75	2.98
8		C	9.75	0.95
9		Shell	9.75	3.3
10		Go	10.5	2.08

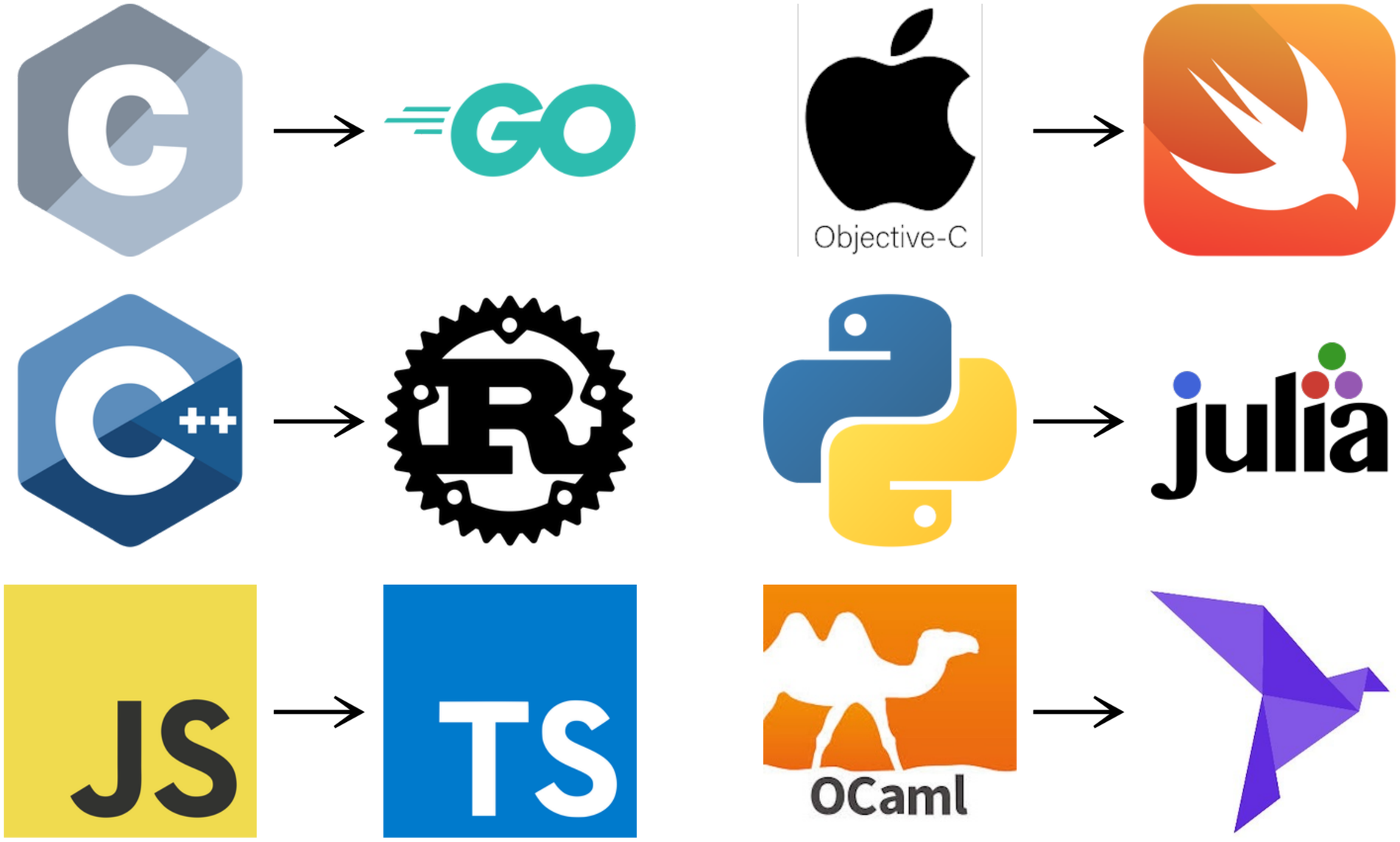
		Language	Avg	StDev
11		Rust	13	2.64
12		Kotlin	13.66	1.52
13		Swift	15.66	5.13
14		R	16.66	5.13
15		Dart	17	2
16		PowerShell	17	7
17		Ruby	17	8.18
18		Lua	21	7.07
19		Scala	23	9.84
20		Objective-C	25	10.58

Modern not Contemporary

Efficient not Fast

Highlighted not Complete

Evolution not History


C
GO
Objective-C
C++
R
julia
JS
TS
OCaml

Execution Model

Compiled Languages

Prioritize Performance

C/C++ Compilation Model

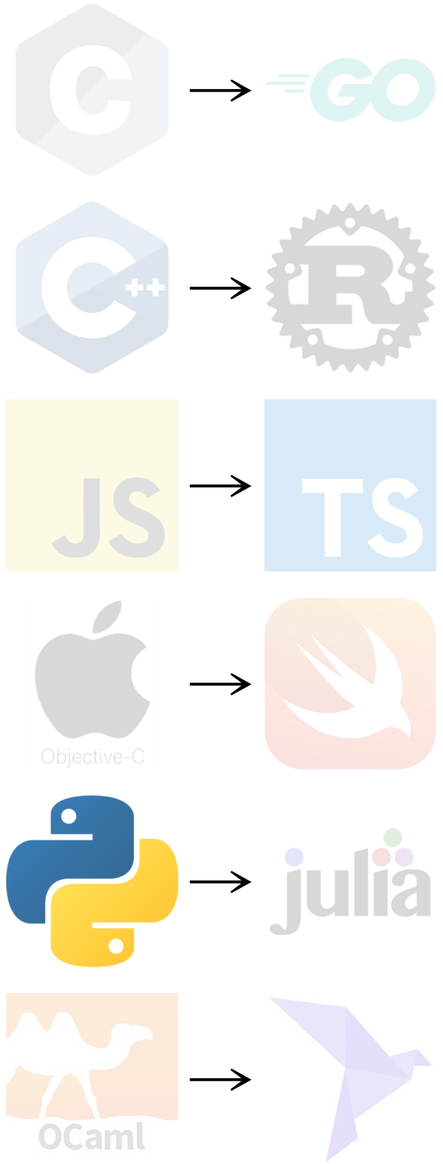
Run on Bear Metal

Interpreted Languages

Prioritize Interactivity

Require a RunTime

Dying Breed?

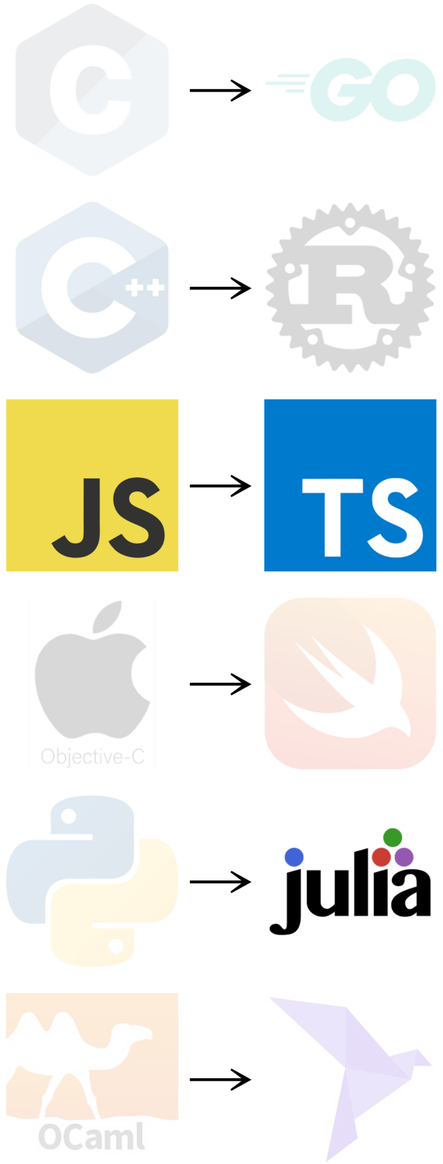


Jitted Languages

Performance-and-Interactivity

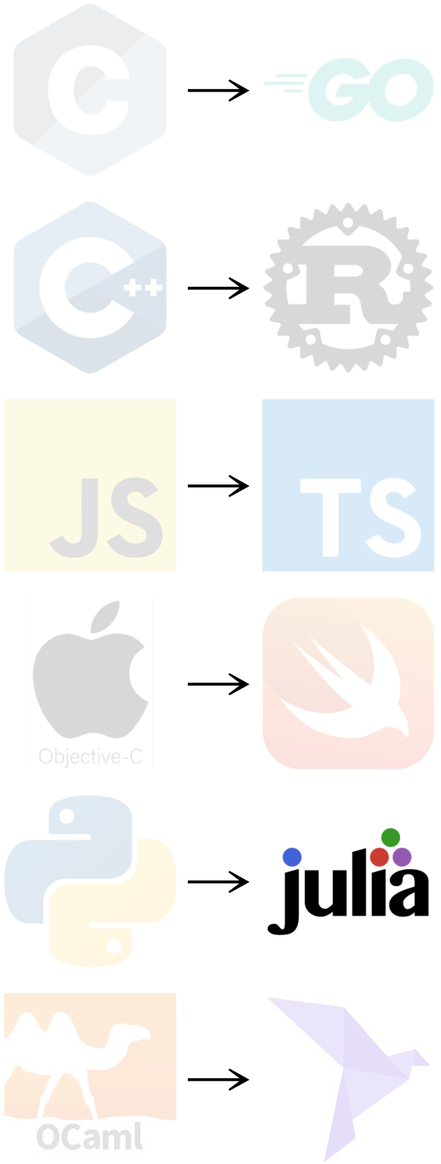
Require a RunTime

Different Jitting Strategies



Compile on Demand

```
julia> f(x) = 1 + x
f (generic function with 1 method)

julia> typeof(f).name.mt.defs.func.specializations
svec()

julia> f(1)
2

julia> typeof(f).name.mt.defs.func.specializations
MethodInstance for f(::Int64)

julia> f(3.2)
4.2

julia> typeof(f).name.mt.defs.func.specializations
svec(MethodInstance for f(::Int64), MethodInstance for f(::Float64),
```



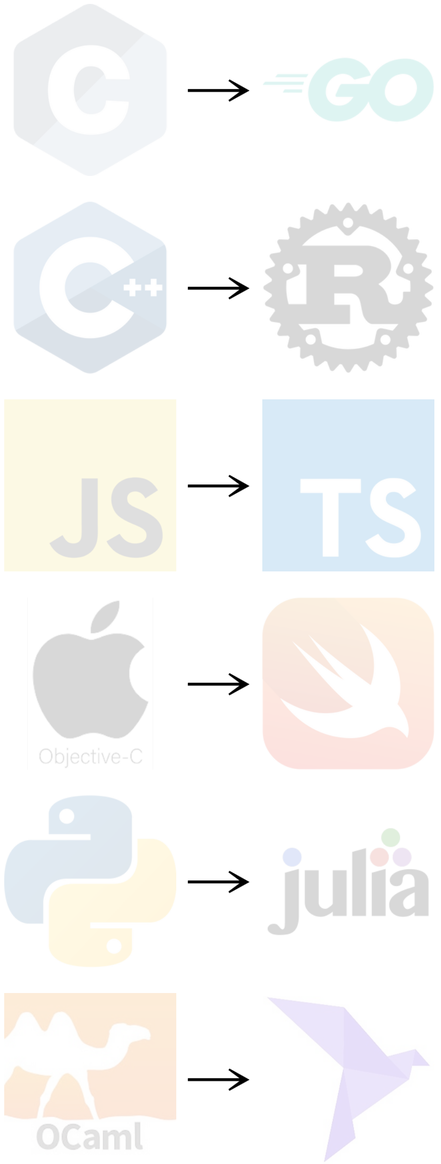
Compilation Strategy

Prioritize Efficiency

No new Interpreters

Interactivity via Jitting

Performance via AOT



Memory Model

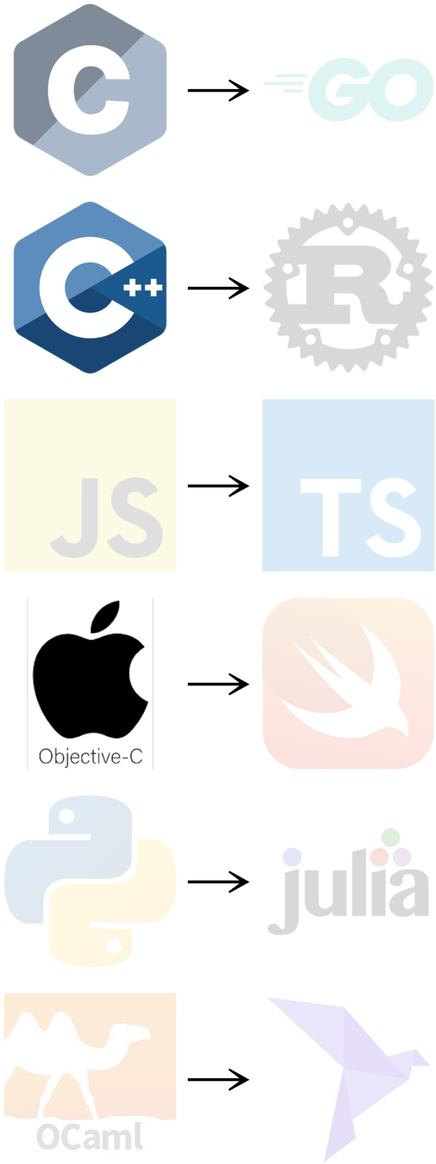
Manual Memory Management

Prioritize Performance

Safety Concerns

Predictable Performance

Run on Bare Metal



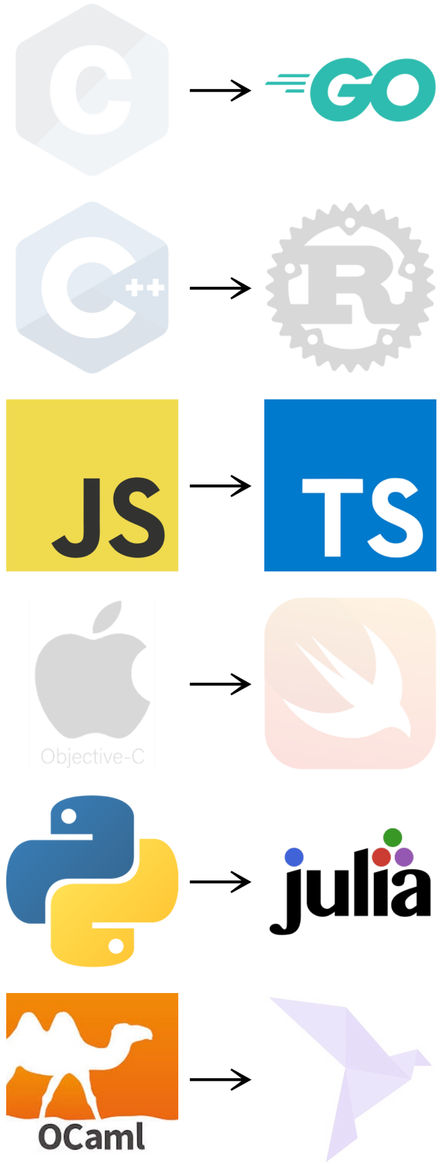
Garbage Collection

Prioritize Safety

Performance Overhead

Unpredictable Performance

Require a RunTime



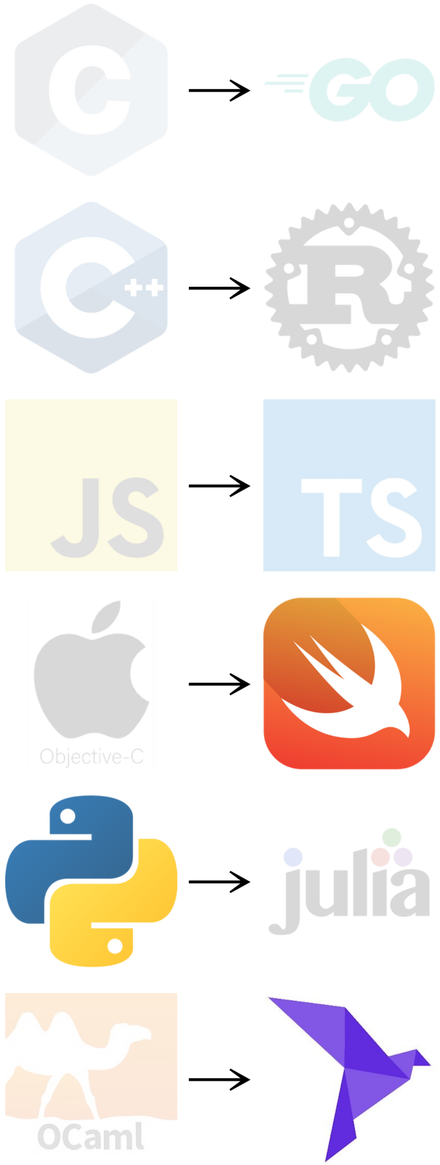
Reference Counting

Prioritize Safety

Performance Overhead

Predictable Performance

Run on Bare Metal

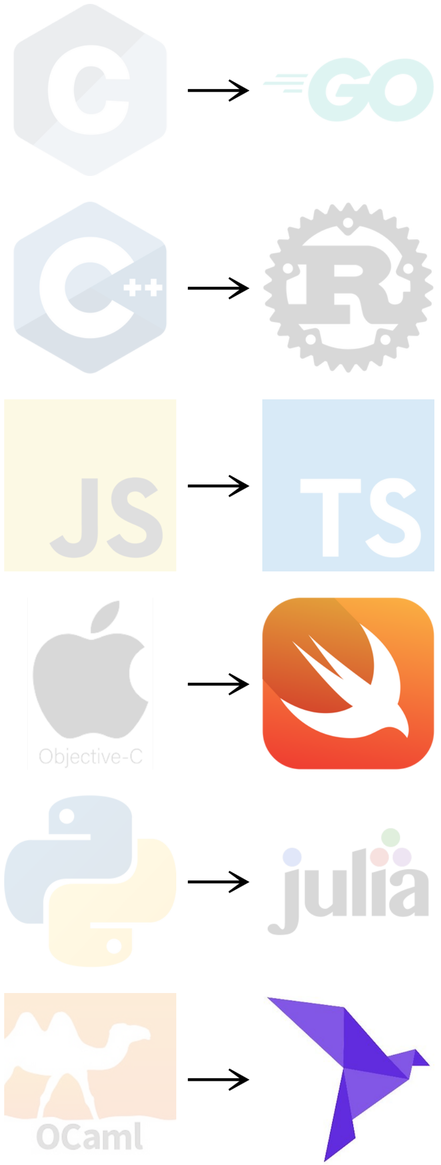


Rc-vs-Gc Debate

GC is generally faster

RC is predictable w/no RT

Reduce RC overhead



Rc-vs-Gc Debate

Perceus: Garbage Free Reference Counting with Reuse

Alex Reinking*
Microsoft Research
Redmond, WA, USA
alex_reinking@berkeley.edu

Ningning Xie*
University of Hong Kong
Hong Kong, China
nnxie@cs.hku.hk

Leonardo de Moura
Microsoft Research
Redmond, WA, USA
leonardo@microsoft.com

Daan Leijen
Microsoft Research
Redmond, WA, USA
daan@microsoft.com

Abstract

We introduce Perceus, an algorithm for precise reference counting with reuse and specialization. Starting from a functional core language with explicit control-flow, Perceus emits precise reference counting instructions such that (cycle-free) programs are *garbage free*, where only live references are retained. This enables further optimizations, like reuse analysis that allows for guaranteed in-place updates at runtime. This in turn enables a novel programming paradigm that we call *functional but in-place* (FBIP). Much like tail-call optimization enables writing loops with regular function calls, reuse analysis enables writing in-place mutating algorithms in a purely functional way. We give a novel formalization of reference counting in a linear resource calculus, and prove that Perceus is sound and garbage free. We show evidence that Perceus, as implemented in Koka, has good performance and is competitive with other state-of-the-art memory collectors.

CCS Concepts: • **Software and its engineering → Runtime environments**; **Garbage collection**; • **Theory of computation → Linear logic**.

Keywords: Reference Counting, Algebraic Effects, Handlers

ACM Reference Format:
Alex Reinking, Ningning Xie, Leonardo de Moura, and Daan Leijen. 2021. Perceus: Garbage Free Reference Counting with Reuse. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3453483.3454032

1 Introduction

Reference counting [7], with its low memory overhead and ease of implementation, used to be a popular technique for automatic memory management. However, the field has broadly moved in favor of generational tracing collectors [31], partly due to various limitations of reference counting, including cycle collection, multi-threaded operations, and expensive in-place updates.

In this work we take a fresh look at reference counting. We consider a programming language design that gives strong compile-time guarantees in order to enable efficient reference counting at run-time. In particular, we build on the pioneering reference counting work in the Lean theorem prover [46], but we view it through the lens of language design, rather than purely as an implementation technique.

We demonstrate our approach in the Koka language [23, 25]: a functional language with mostly immutable data types together with a strong type and effect system. In contrast to the dependently typed Lean language, Koka is general-purpose, with support for exceptions, side effects, and mutable references via general algebraic effects and handlers [39, 40]. Using recent work on evidence translation [50–52], all these control effects are compiled into an internal core language with explicit control flow. Starting from this functional core, we can statically transform the code to enable efficient reference counting at runtime. In particular:

- Due to explicit control flow, the compiler can emit *precise* reference counting instructions where a (non-cyclic) reference is dropped as soon as possible. We call this *garbage free* reference counting as only live data is retained (§ 2.2).

FP^2: Fully in-Place Functional Programming

ANTON LORENZEN, University of Edinburgh, UK
DAAN LEIJEN, Microsoft Research, USA
WOUTER SWIERSTRA, Universiteit Utrecht, Netherlands

As functional programmers we always face a dilemma: should we write purely functional code, or sacrifice purity for efficiency and resort to in-place updates? This paper identifies precisely when we can have the best of both worlds: a wide class of purely functional programs can be executed safely using in-place updates without requiring allocation, provided their arguments are not shared elsewhere.

We describe a linear *fully in-place* (FIP) calculus where we prove that we can always execute such functions in a way that requires no (de)allocation and uses constant stack space. Of course, such a calculus is only relevant if we can express interesting algorithms; we provide numerous examples of in-place functions on datastructures such as splay trees or finger trees, together with in-place versions of merge sort and quick sort.

We also show how we can generically derive a map function over *any* polynomial data type that is fully in-place. Finally, we have implemented the rules of the FIP calculus in the Koka language. Using the Perceus reference counting garbage collection, this implementation dynamically executes FIP functions in-place whenever possible.

CCS Concepts: • **Software and its engineering → Control structures**; *Recursion*; • **Theory of computation → Operational semantics**.

Additional Key Words and Phrases: FBIP, Tail Recursion Modulo Cons

ACM Reference Format:
Anton Lorenzen, Daan Leijen, and Wouter Swierstra. 2023. FP^2: Fully in-Place Functional Programming. *Proc. ACM Program. Lang.* 7, ICFP, Article 198 (August 2023), 30 pages. https://doi.org/10.1145/3607840

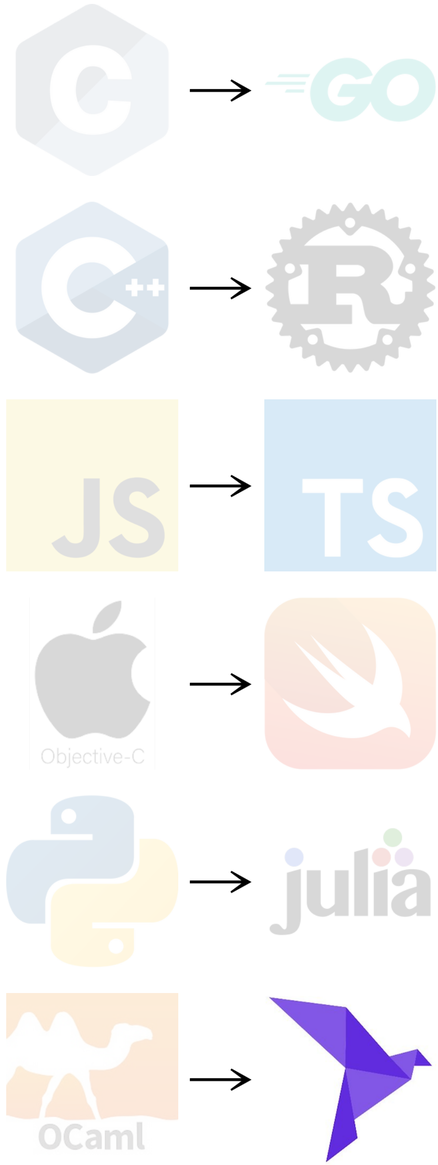
1 INTRODUCTION AND OVERVIEW

The functional program for reversing a list in linear time using an accumulating parameter has been known for decades, dating back at least as far as Hughes's work on difference lists [1986]:

```
fun reverse-acc( xs : list<a>, acc : list<a> ) : list<a>
  match xs
    Cons(x,xx) -> reverse-acc( xx, Cons(x,acc) )
    Nil        -> acc

fun reverse( xs : list<a> ) : list<a>
  reverse-acc(xs,Nil)
```

As this definition is *pure*, we can calculate with it using equational reasoning in the style of Bird and Meertens [Backhouse 1988; Gibbons 1994]. Using simple induction, we can, for instance, prove that this linear time list reversal produces the same results as its naive quadratic counterpart.

Not all in the garden is rosy: what about the function's memory usage? The purely functional definition of `reverse` allocates fresh `Cons` nodes in each iteration; an automatic garbage collector needs to discard unused memory. This generally induces a performance penalty relative to an imperative



Mostly Heap

Most Objects on the Heap

Optimization for Scalars

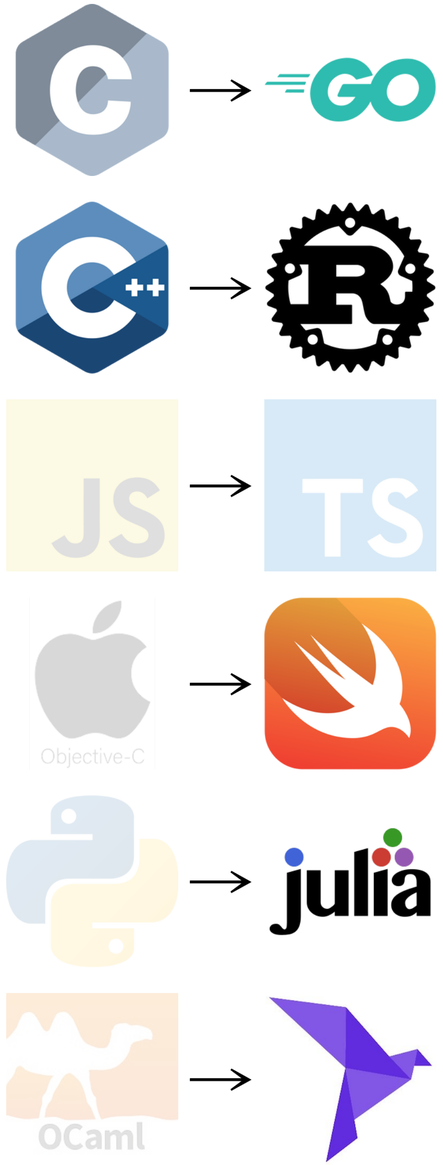
Increases GC/RC pressure

Inline and Heap

Objects may be Inlined

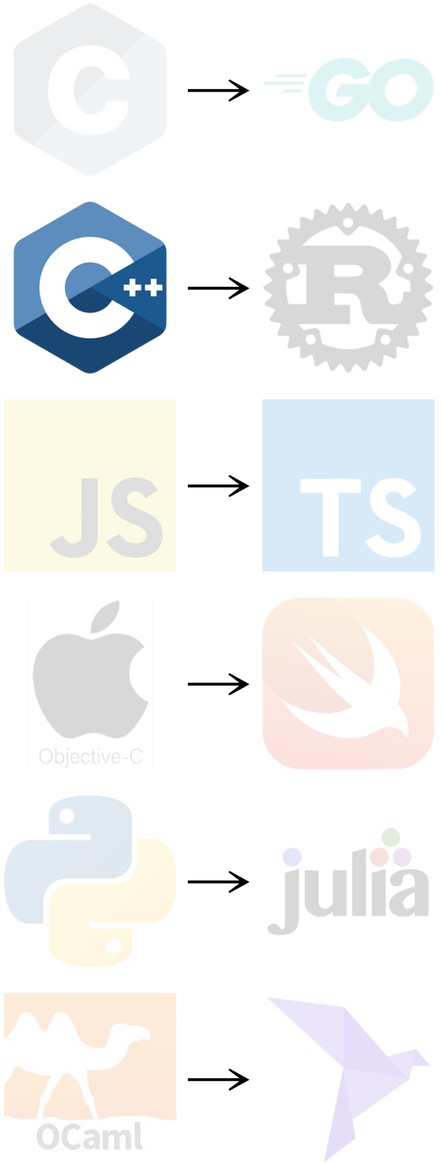
Per-type or per-instance

Reduced GC/RC pressure



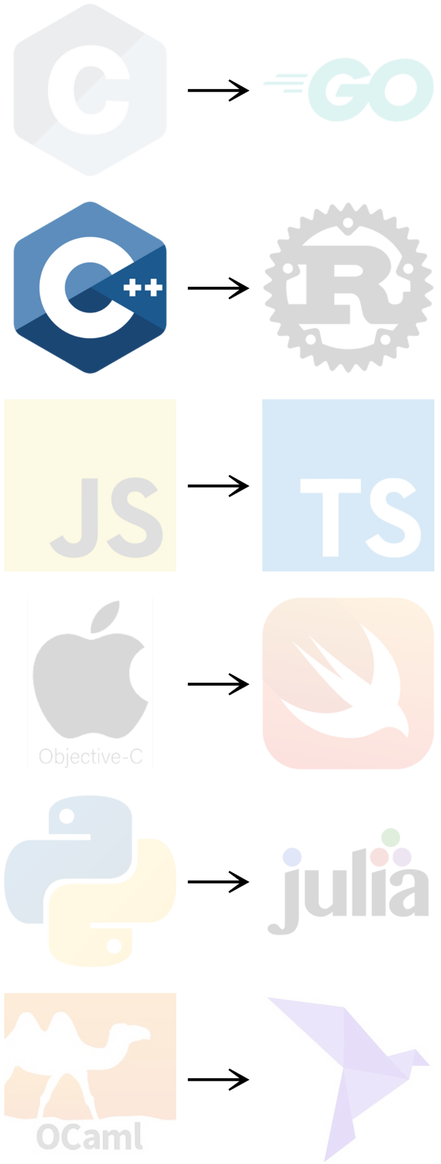
Memory Safety Issues in Cpp

```cpp
void function() {
  // manual memory management - safety issues
  auto optr = new int{42}; use(optr); delete optr;
  // automatic memory management - single ownership
  auto uptr = std::make_unique<int>(42); use(uptr);
  // automatic memory management - shared ownership
  auto sptr = std::make_shared<int>(42); use(sptr);

  // manual memory management - safety issues
  auto aptr = new int[100]; use(aptr); delete[] aptr;
  // automatic memory management - value type
  auto vval = std::vector<int>(100); use(vval);
}
```



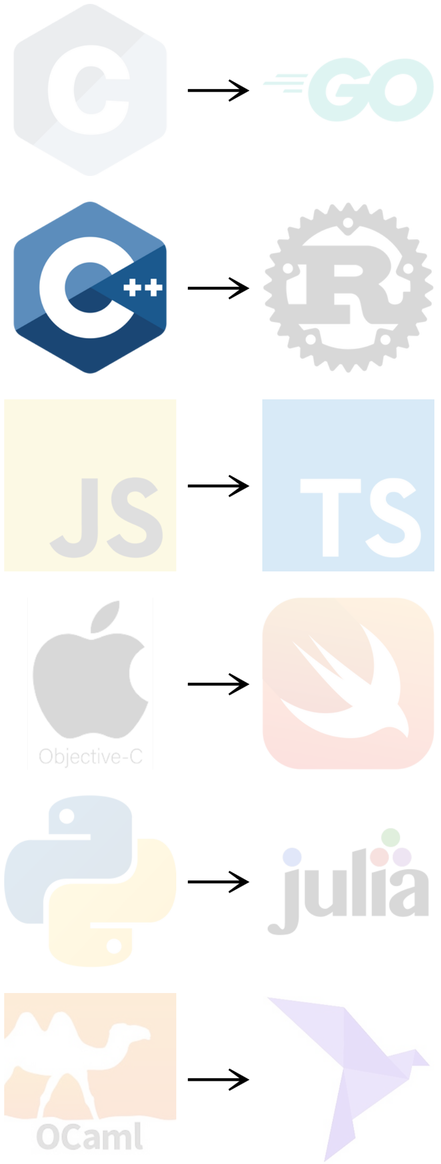
Memory Safety Issues in Cpp

```cpp
void function() {
    // manual memory management - safety issues
    auto optr = new int{42}; use(optr); delete optr;
    // automatic memory management - single ownership
    auto uptr = std::make_unique<int>(42); use(uptr);
    // automatic memory management - shared ownership
    auto sptr = std::make_shared<int>(42); use(sptr);

    // manual memory management - safety issues
    auto aptr = new int[100]; use(aptr); delete[] aptr;
    // automatic memory management - value type
    auto vval = std::vector<int>(100); use(vval);
}
```



Memory Safety Issues in Cpp

```cpp
auto issue1() { // escaped local reference
  auto value = 42; return &value; }
auto issue2() { // escaped in lambda
  auto value = 42; return [&]{ return value; }; }
auto issue3() { // invalid view
  auto vec = std::vector<int>{1};
  return std::span{vec}; }
auto issue4() { // reference invalidated
  auto vec = std::vector<int>{1};
  auto& ref = vec[0]; vec.resize(10); }
```



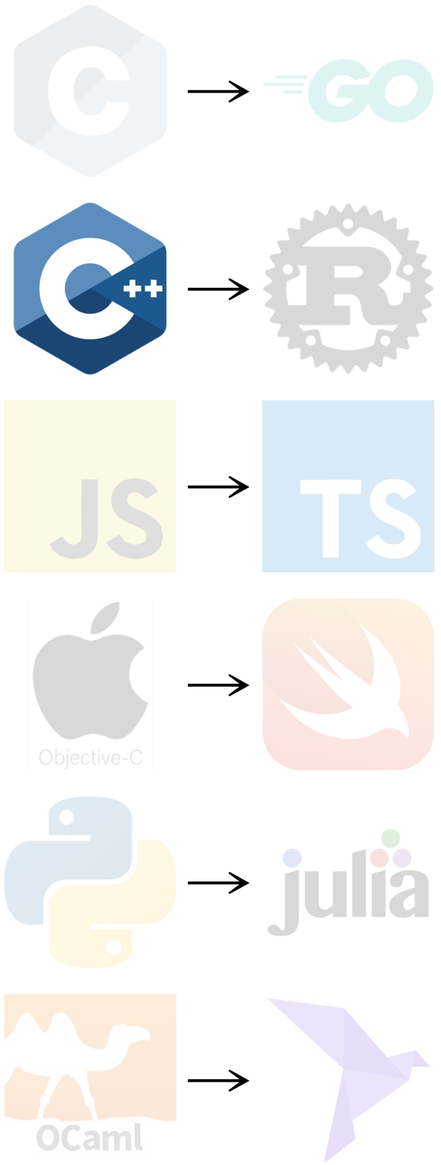
Memory Safety Issues in Cpp

Expose pointers

Makes no ownership assumption

Ranges model pointers

Pointer access is unsafe

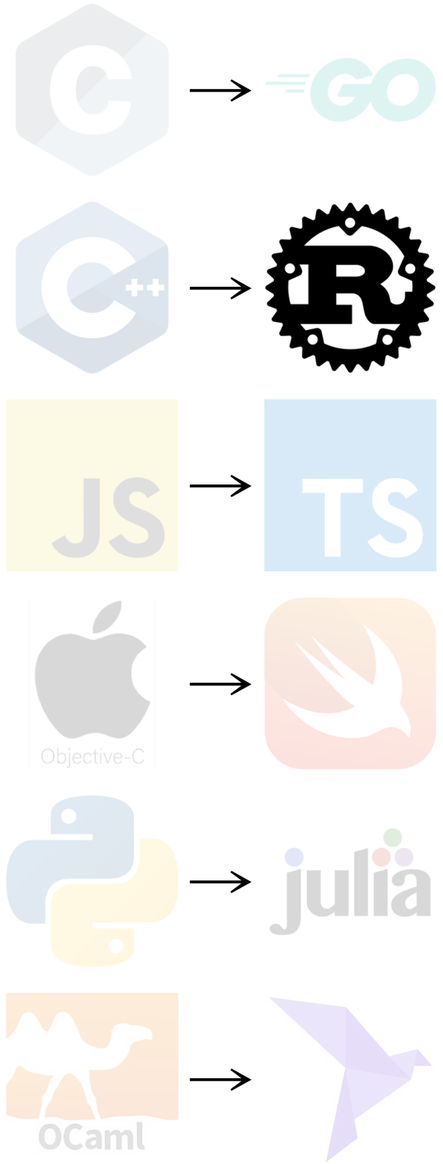


Rust Memory Management

Similar references as C++

Favors single ownership

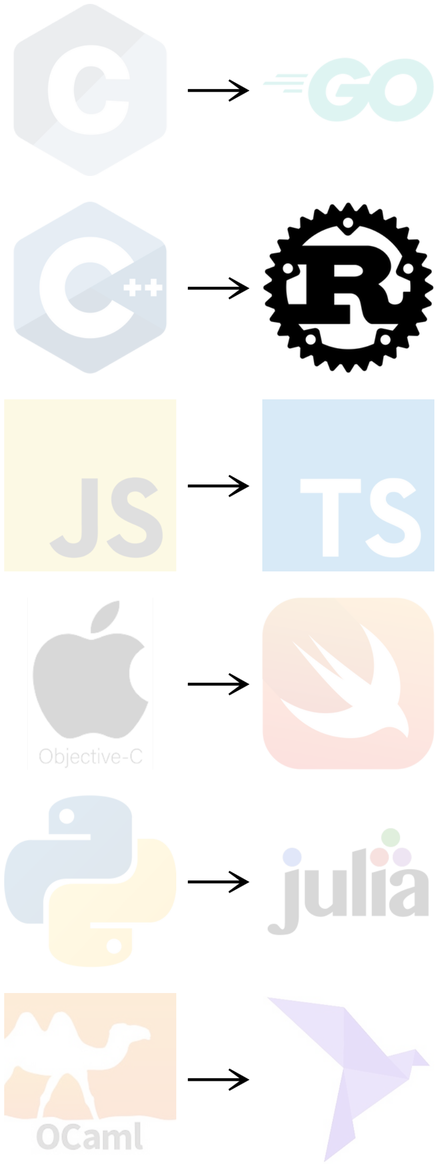
Verified at compiled time



Rust Memory Management

```rust
fn function1() {
    let value = 42; let reference = &value;
    let bad_ref = { let value = 42; &value };
}
```

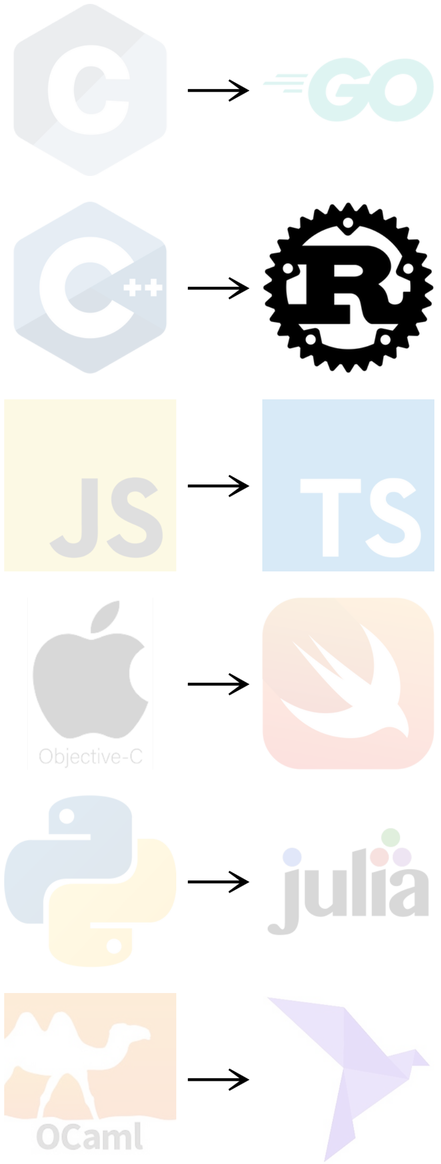
```
error[E0597]: `value` does not live long enough
 --> src/main.rs:8:37
  |
8 |     let bad_ref = { let value = 42; &value };
  |         -------          -----      ^^^^^^ - `value` dropped here while still borrowed
  |         |                |          |
  |         |                |          borrowed value does not live long enough
  |         |                binding `value` declared here
  |         borrow later stored here
```



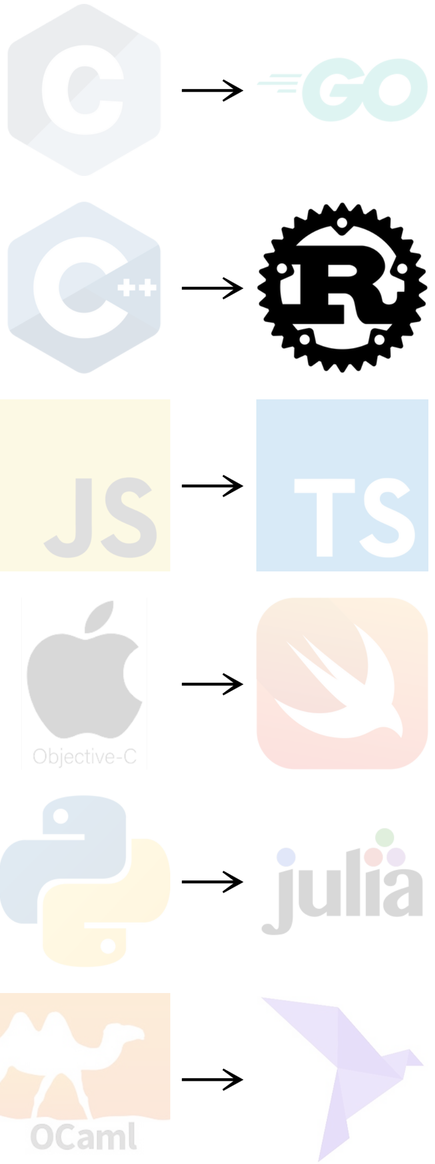
Rust Memory Management

```rust
fn function2() {
    let mut values = vec![1, 2, 3];
    let reference = &mut values[0];
    values.push(4); *reference = 0;
}
```

```
error[E0499]: cannot borrow `values` as mutable more than once at a time
  --> src/main.rs:14:5
   |
13 |     let reference = &mut values[0];
   |                           ------ first mutable borrow occurs here
14 |     values.push(4); *reference = 0;
   |     ^^^^^^          ------------ first borrow later used here
   |     |
   |     second mutable borrow occurs here
```



Rust Memory Management



1. One mutable reference or any number of immutable references
2. References must be valid
3. Data must outlive references

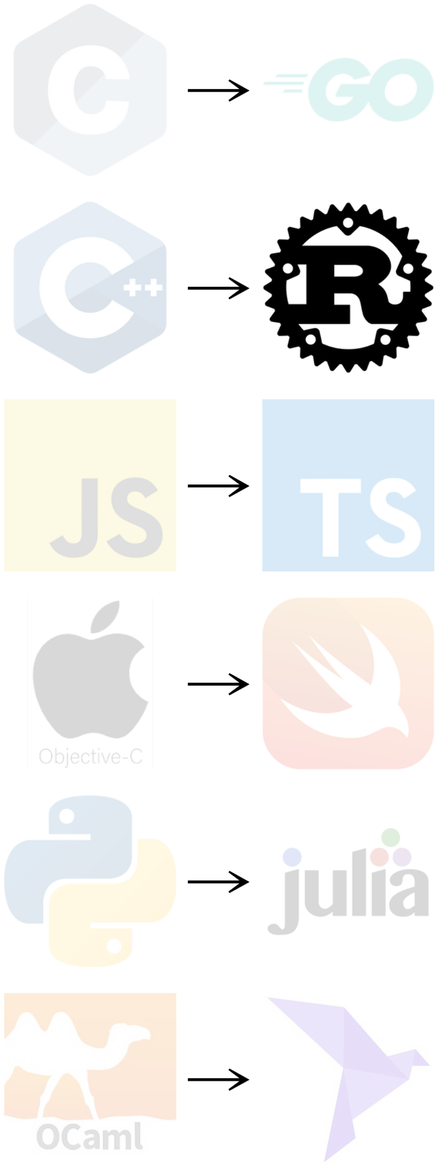
Rust Memory Management

Adds Complexity

Reduced by Type Inference

Trade-off with Safety Guarantees

Unsafe escape hatch



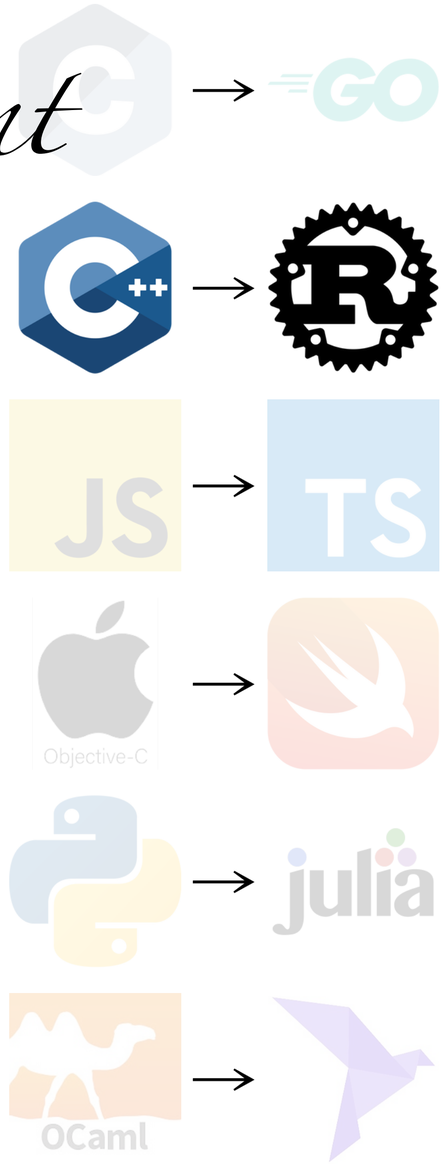
Cpp-vs-Rust Memory Management

Same Memory Model

Rust: Favors Safety

C++: Favors Simplicity

Industry is switching to Rust



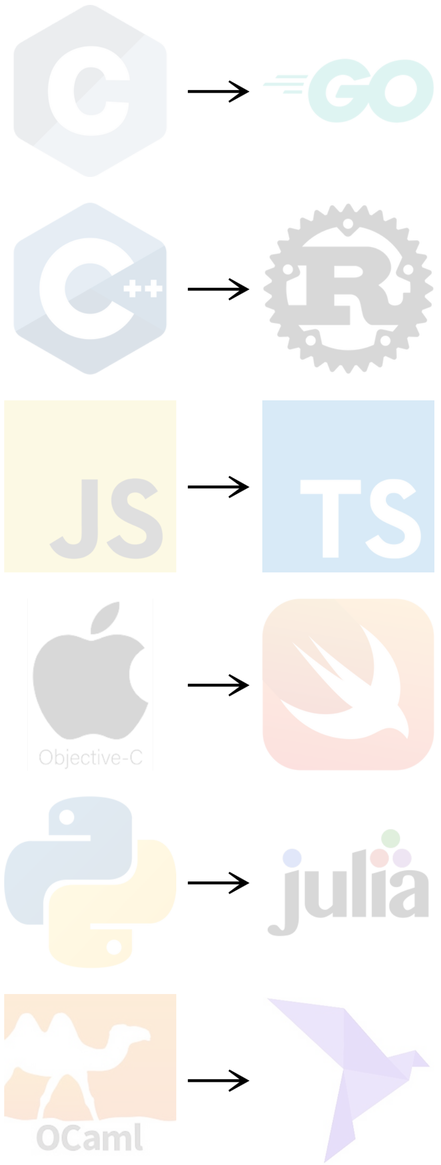
Safe Memory Model

Prioritize Safe Memory

Simple-but-Inefficient

Complex-but-Efficient

Compiler to Reduce Overhead



Object Model

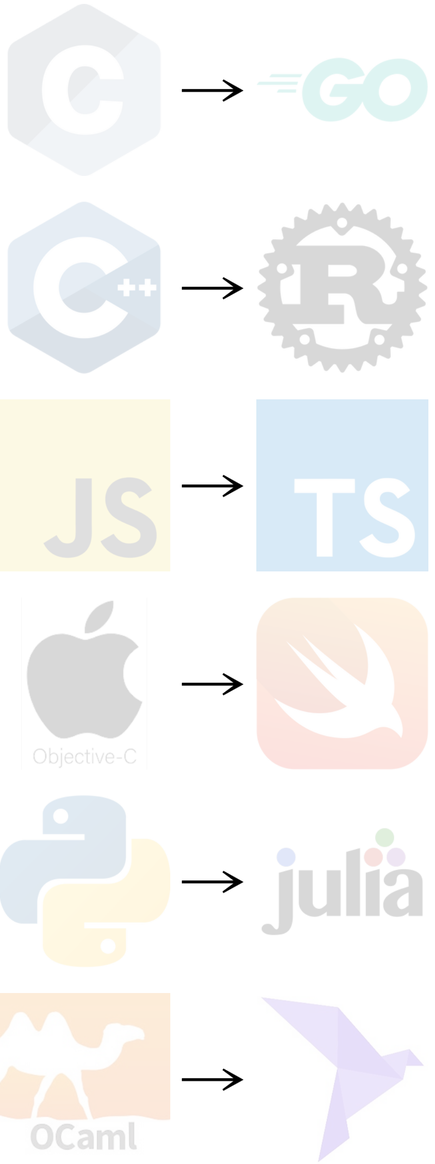
Values, References, Moves

Default for assignment and calls

Values: behave like *int*

References: behave like *int&*

“Moves”: behave like *int&&*

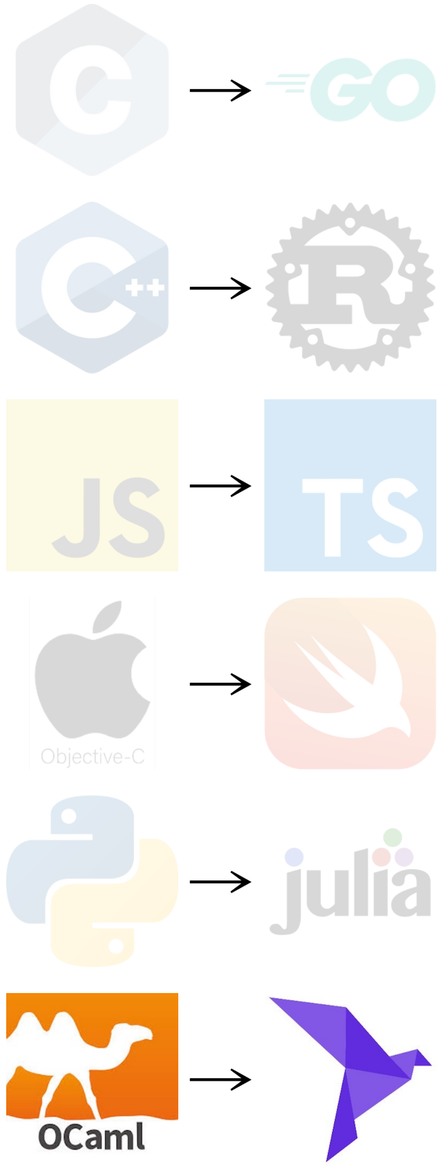


Values-only

Objects are Values

Combined with Immutability

Referential Transparency



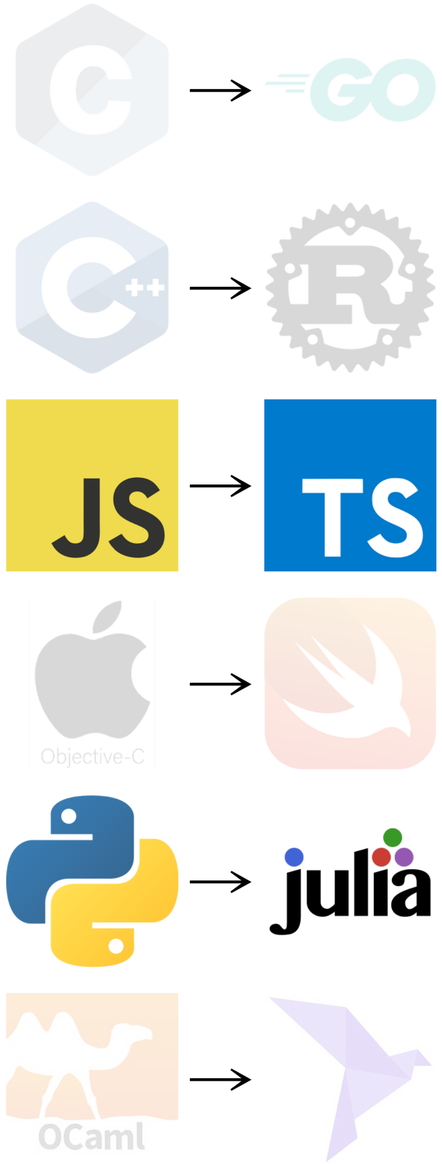
References-only

Objects are References

May Optimize Scalars

Combined with Mutability

Convenient, but error prone



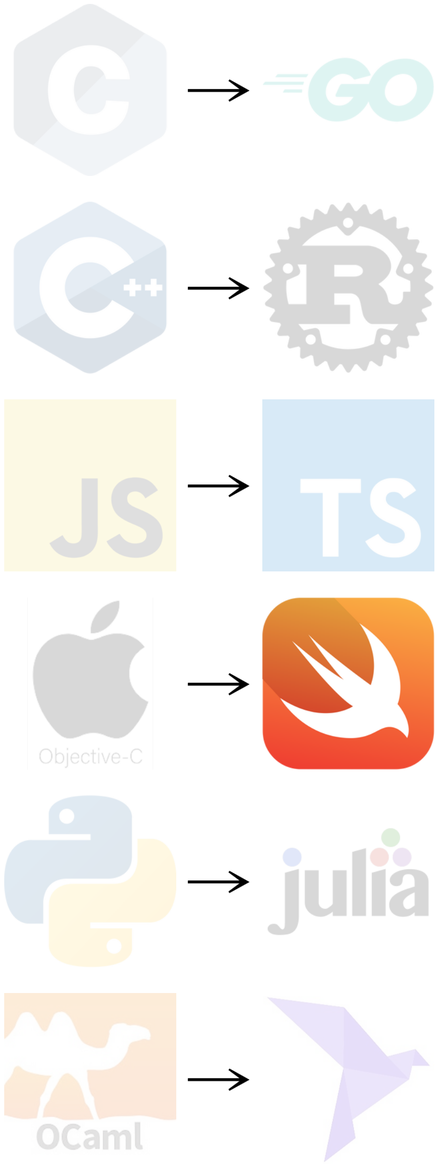
Values and References

Semantic by Types

Prioritize Simplicity

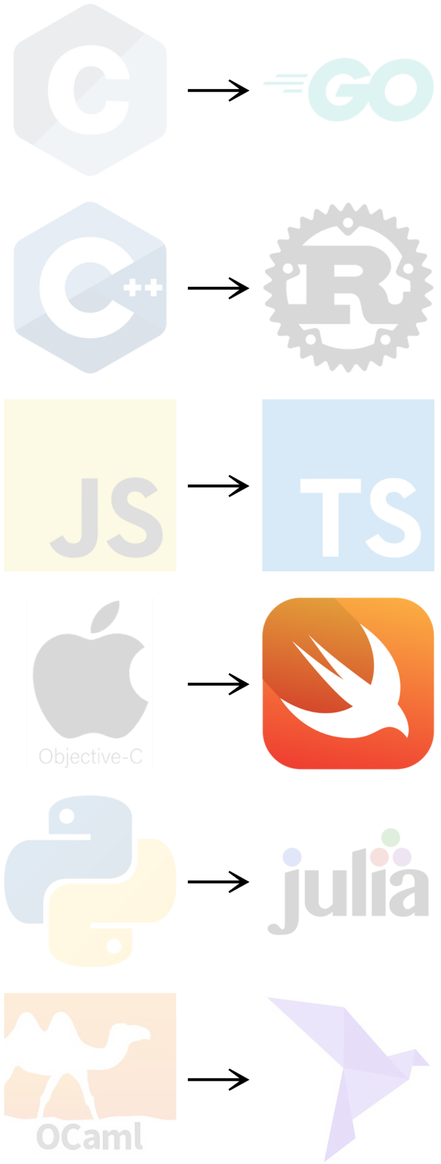
Prefer Values

Eg: Collections are COW Types



Values and References

```
struct Point { // struct is a value type
    var x: Float
    var y: Float
}
class Shape { // class is a reference type
    var pos: Point
    var col: String
}
let p = Point(x: 1.0, y: 2.0)      // value
let s = Shape(pos: p, col: "red") // reference
```



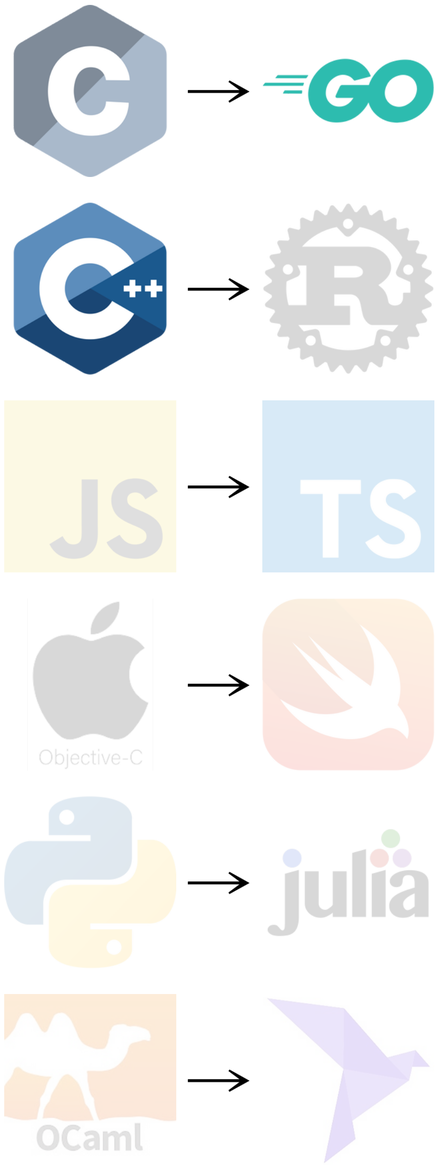
Values-default

Explicit References

Prioritize Control

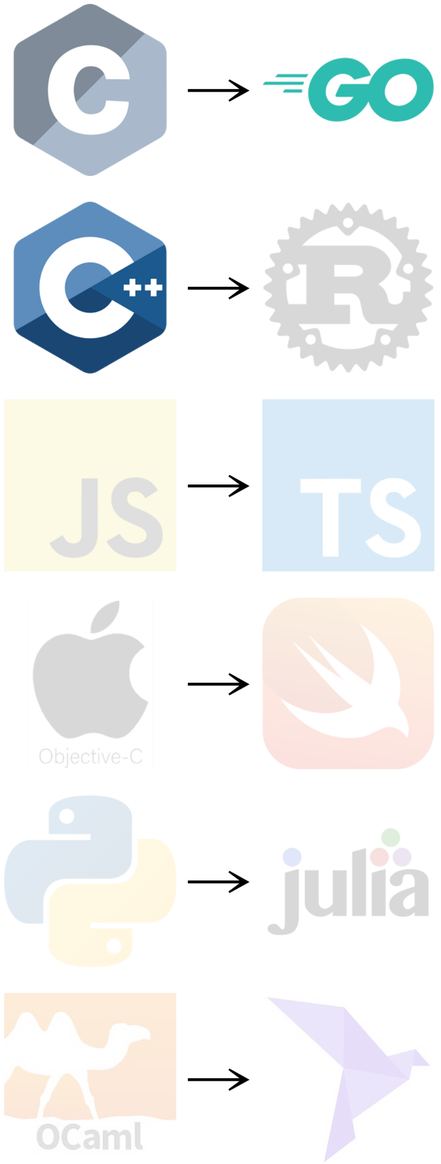
Refs for Efficient and Sharing

Prefer Values



Values-default

```
void values() {
    auto sv = 1;                // value
    auto sc = sv;               // copy
    auto sr = &sv;              // reference
    auto vv = vector{1, 2}; // value
    auto vc = vv;               // copy
    auto vr = &vv;              // reference
}
```



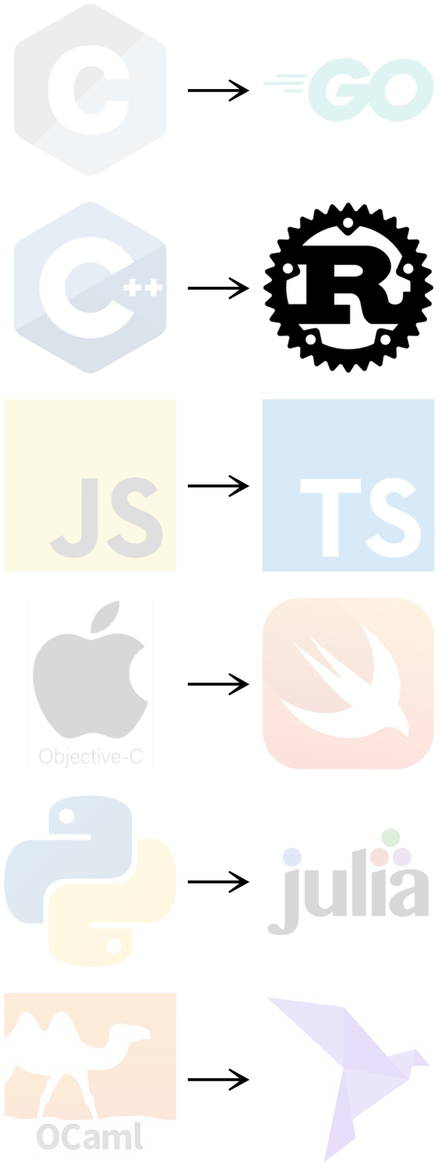
Move-default

Defaults to Destructive Moves

Explicit Copies

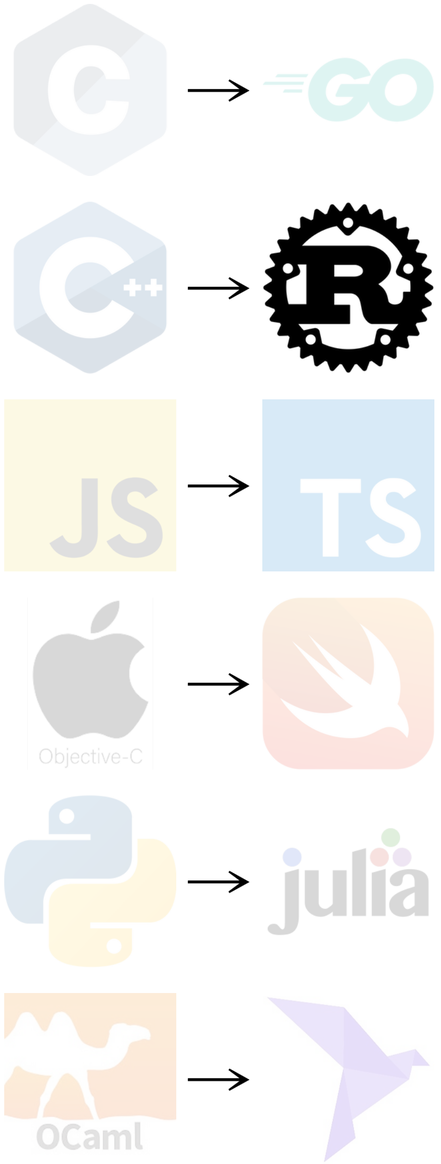
Explicit Safe References

Mutability discouraged



Move-default

```rust
fn moves() {
    let _sv: i32 = 42; // value
    let _sr: &i32 = &_sv; // reference
    let _sc: i32 = _sv; // copy
    let _vv: Vec<i32> = vec![1, 2]; // value
    let _vr: &Vec<i32> = &_vv; // reference
    let _vc: Vec<i32> = _vv; // move
    let _v2: Vec<i32> = _vv; // error
}
```



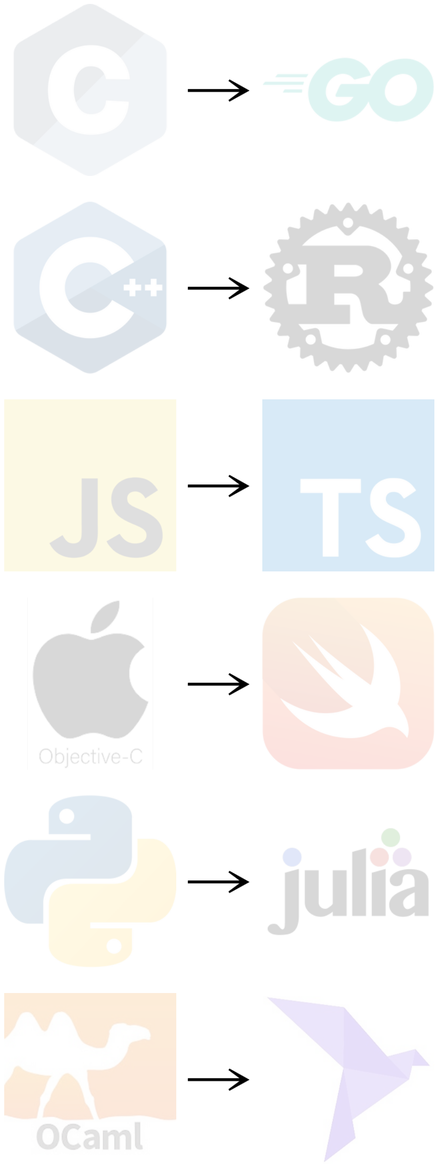
Values-vs-References

Varies wildly in Languages

Backward compatibility

Referential Transparency

Verbose Efficiency



Type System

Dynamic Typing

```
function f(x, y) { return x + y; }
```

```
def f(x, y): return x + y
```

Favors experimentation

Easy to start, hard to maintain

Static Typing

```c
int f(int x, int y) { return x + y; }
```

```rust
fn f(x: i32, y: i32) -> i32 { x + y }
```

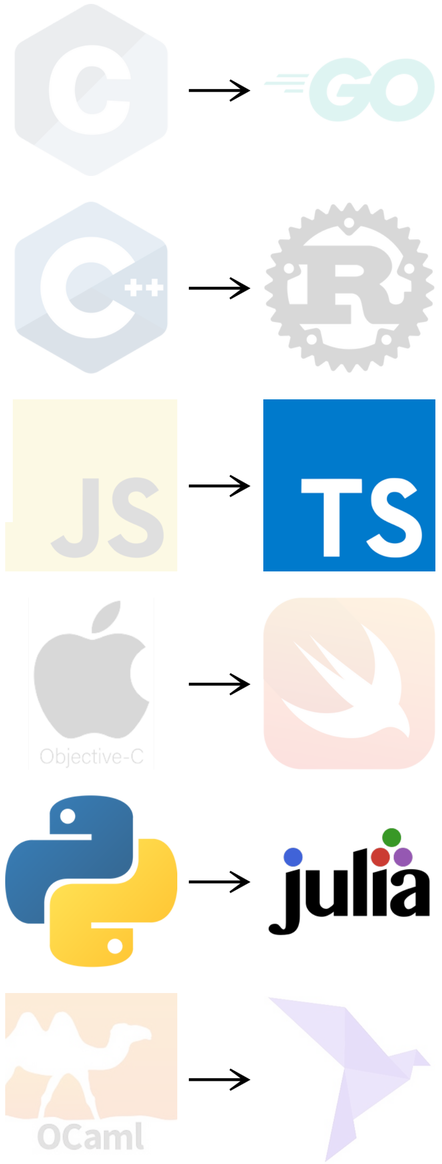
Favors robustness and safety

Hard to start, easier to maintain

Gradual Typing

```
def g(x: int, y: int) -> int: return x + y
```

```
function g(x: number, y: number): number
  { return x + y; }
```



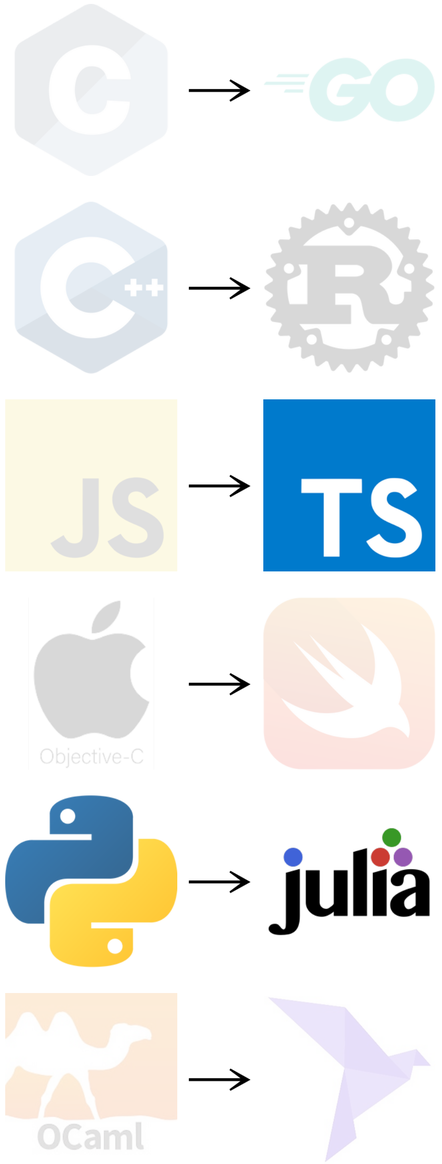
Optional Typing

Start easy, add types later

Gradual Typing - Unsoundness

```ts
function u() {
    const v : number[] = [1, 2, 3];
    const e : number = v[3];
    // e is undefined at runtime
}
```

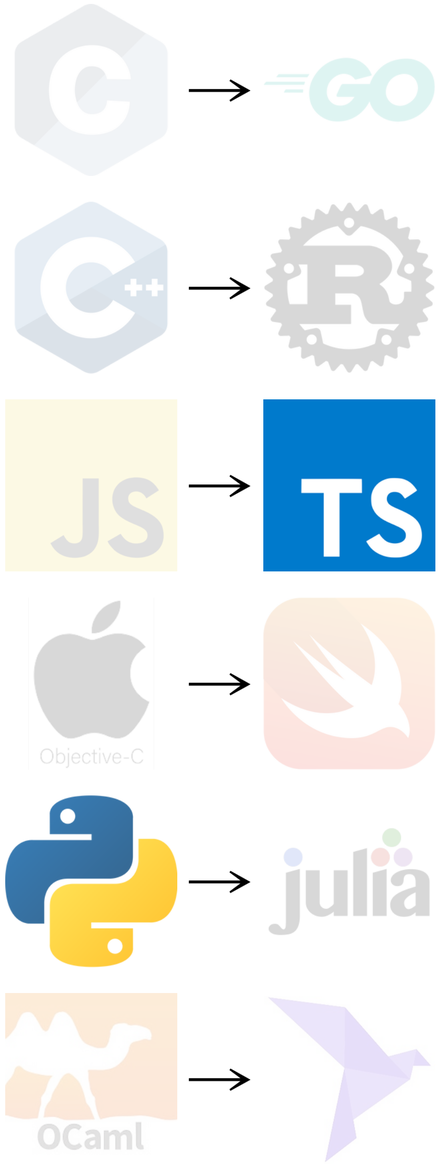
Trade-off for migration



Gradual Typing - Imprecision

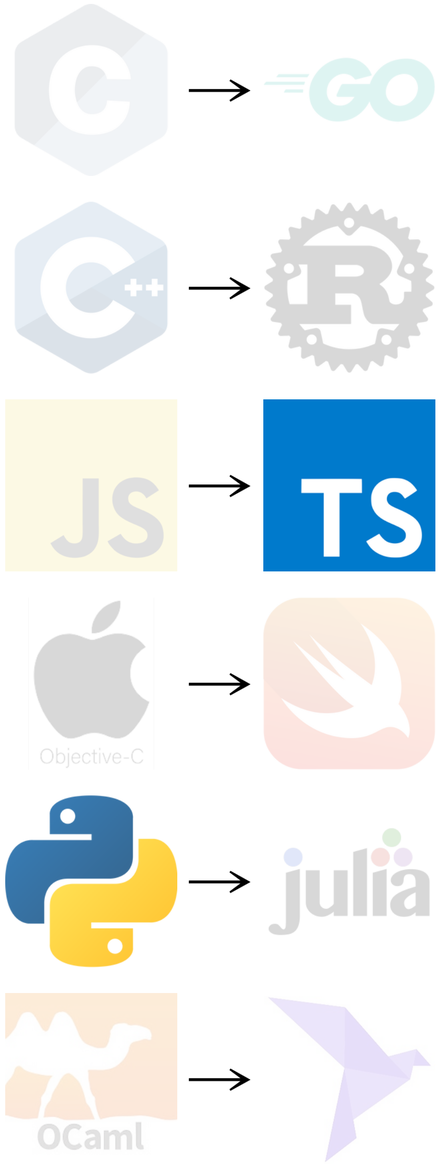
```ts
function g1(x, y: number): number
  { return x + y; }
g1(1, 2);

function g2(x: any, y: number): number
  { return x + y; }
g2(true, 2);
```

Trade-off for migration



Gradual Typing - Imprecision

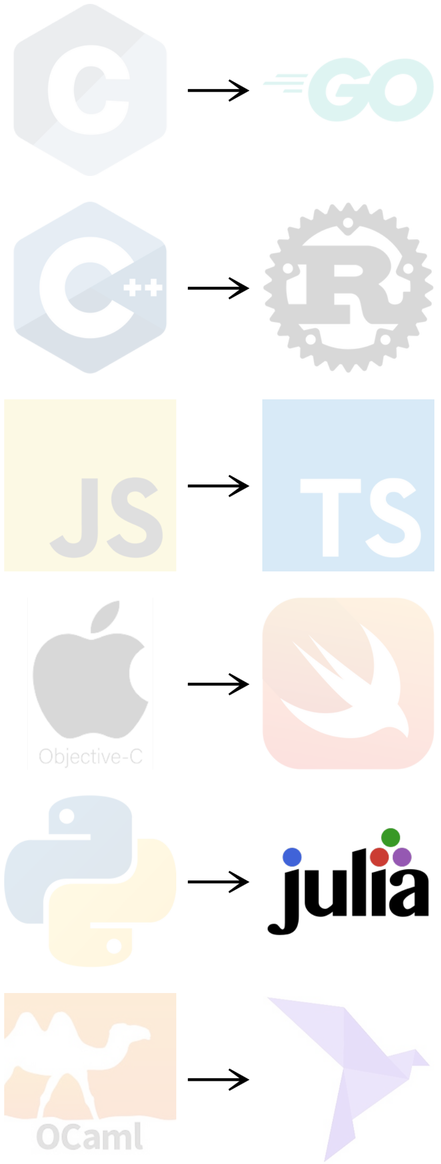
```
def h(a: ArrayLike, b: ArrayLike) → ArrayLike:
    return a + b

h([1, 2], [3, 4])
h(array([1, 2]), array([3, 4]))
h([1, 2], 3)
```



Trade-off for migration

Gradual Typing - Compilation

```
julia> f(x) = 1 + x
f (generic function with 1 method)

julia> typeof(f).name.mt.defs.func.specializations
svec()

julia> f(1)
2

julia> typeof(f).name.mt.defs.func.specializations
MethodInstance for f(::Int64)
```



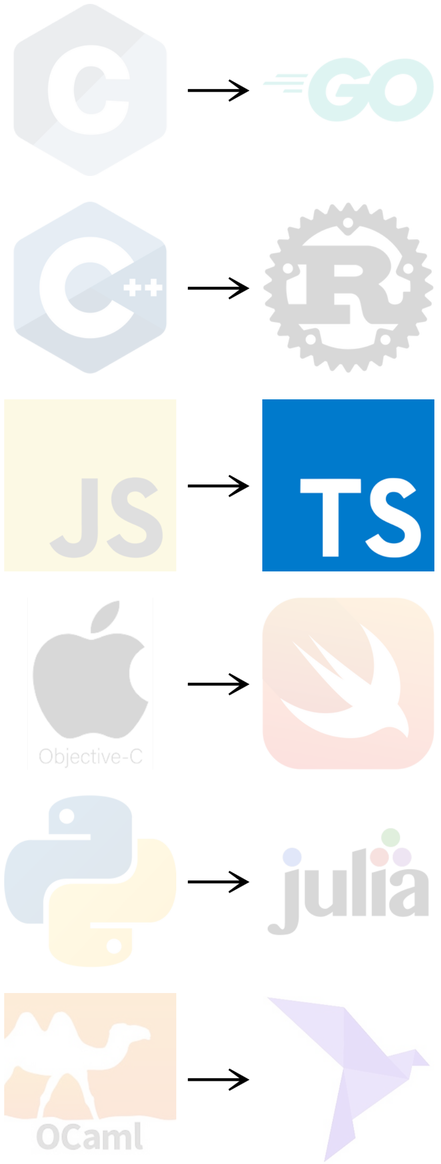
Nominal Types

```
struct Vector2  { float x, y; };
struct Vector2a { float x, y; };

float dot(Vector2 a, Vector2 b) {
  return a.x * b.x + a.y * b.y; }
float length(Vector2a v) {
  return dot(v, v); }
```

Type equality by name

Structural Types

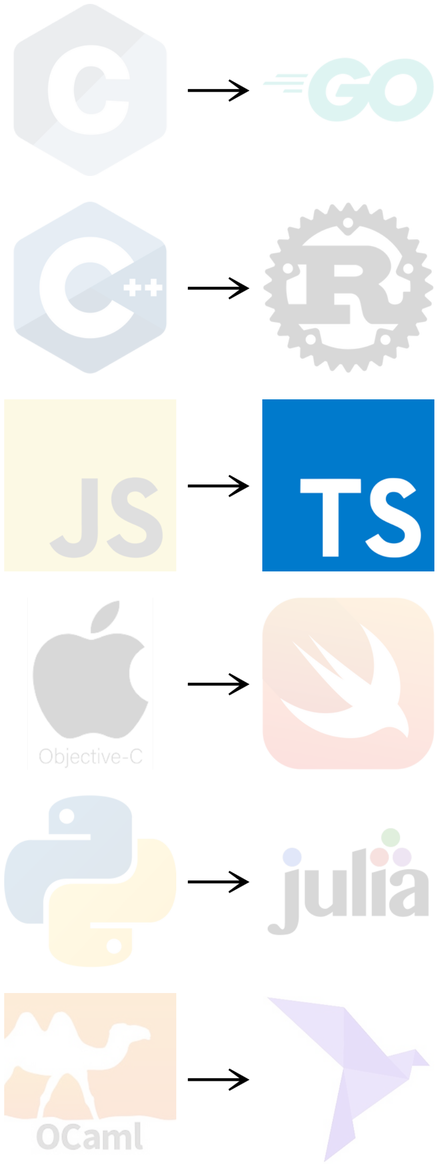
```ts
// product types
interface Vec2 { x: number; y: number; }
function add(p: Vec2, q: Vec2): Vec2 {
  return { x: p.x + q.x, y: p.y + q.y };
}
add({x: 1, y: 2}, {x: 3, y: 4});
```



Type equality based on structure

Structural Types

```ts
// sum types without shared fields
function logme(x: number | string): void {
  console.log(x);
}
logme(42);
logme("hello");
```

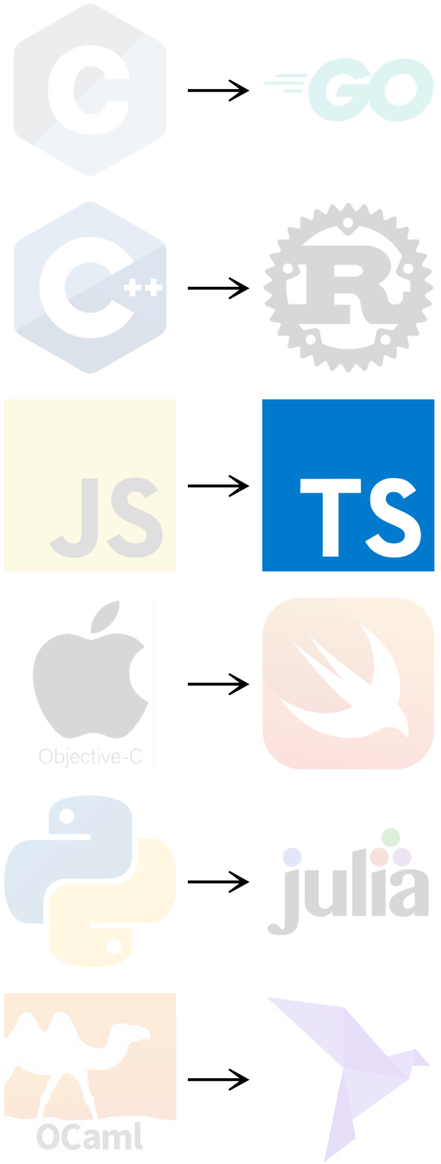


Type equality based on structure

Structural Types

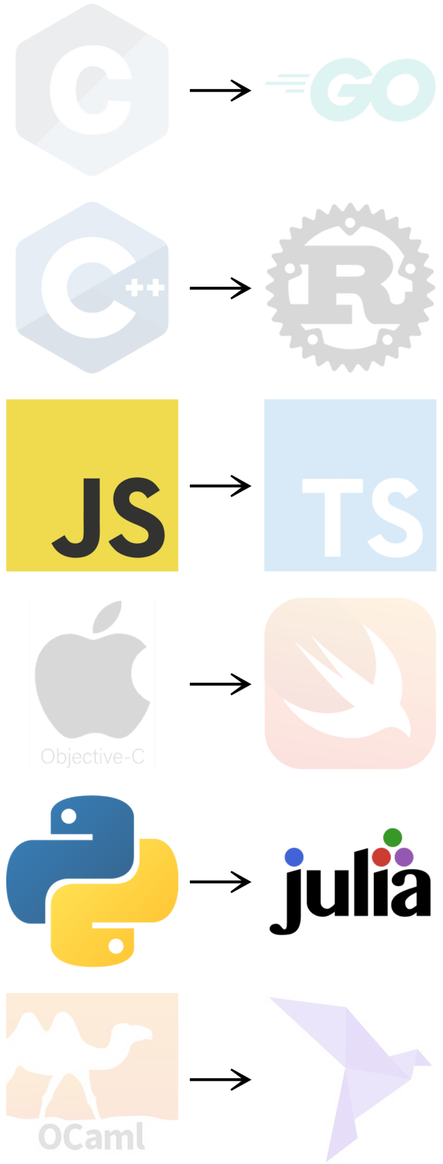
```ts
// sum types with shared fields
interface Circle { kind: "circle"; radius: number; }
interface Square { kind: "square"; side: number; }
function area(s: Circle | Square): number {
  switch (s.kind) {
    case "circle": return Math.PI * s.radius ** 2;
    case "square": return s.side ** 2;
  }
}
```

Type equality based on structure



"Duck" Types

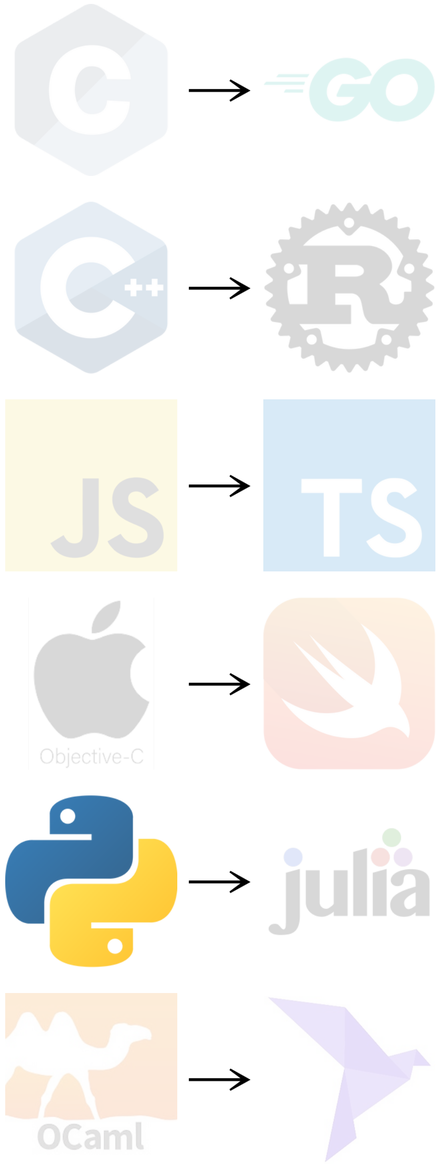
```
def dot(v1, v2) → float:
    return v1[0] * v2[0] + v1[1] * v2[1]

dot([1, 2], [3, 4])
dot((1, 2), (3, 4))
```



Type errors on use

"Duck" Types

```python
Vec2Like = tuple[float, float] | list[float]

def dot(v1: Vec2Like, v2: Vec2Like) -> float:
    return v1[0] * v2[0] + v1[1] * v2[1]
```



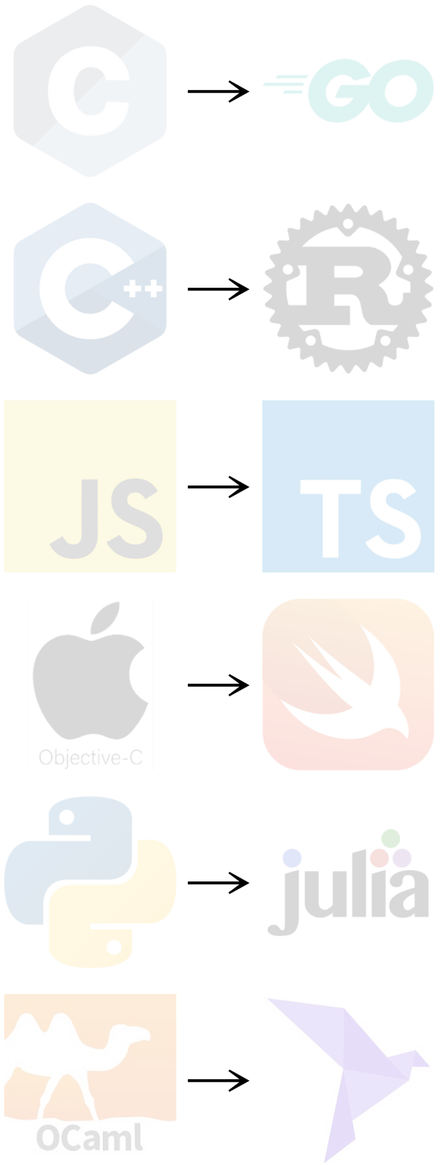
Type errors on use

Static and Nominal

Prioritize Type Safety

Nominal Types Preferred

Gradual Structural Types



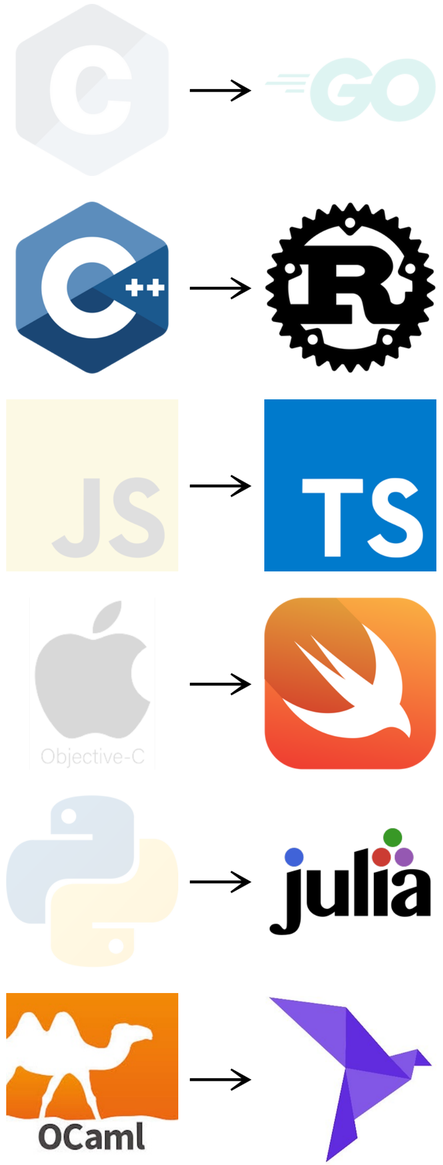
Type Inference

Type Inference

Assign Types Automatically

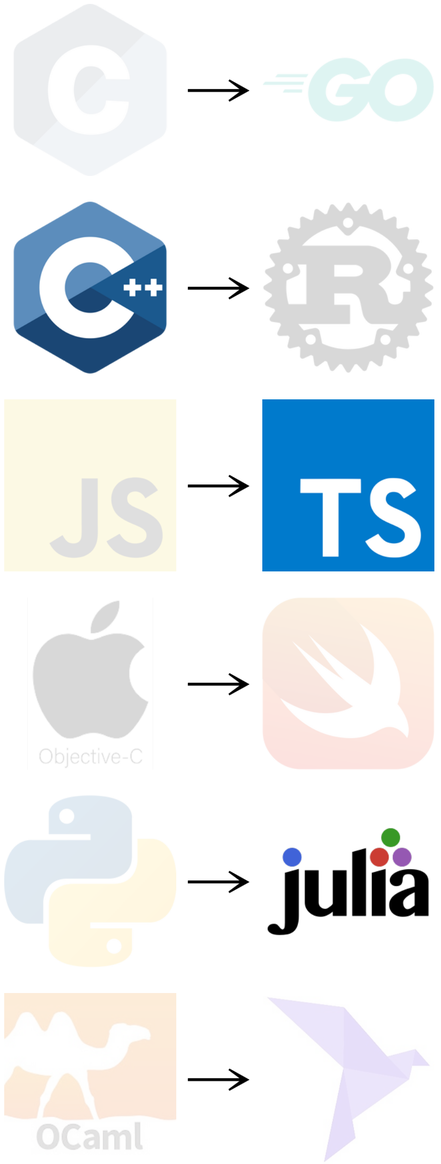
Reduce Typing Overhead

Needed for Generics



Local Type Inference

```ts
function inference() {
  const sv: number = 3;
  const iv : number = sv + 2;
  const av : number[] = [1, 2, 3];
  const bv : number[] = av.map(x ⇒ x + 1);
}
```

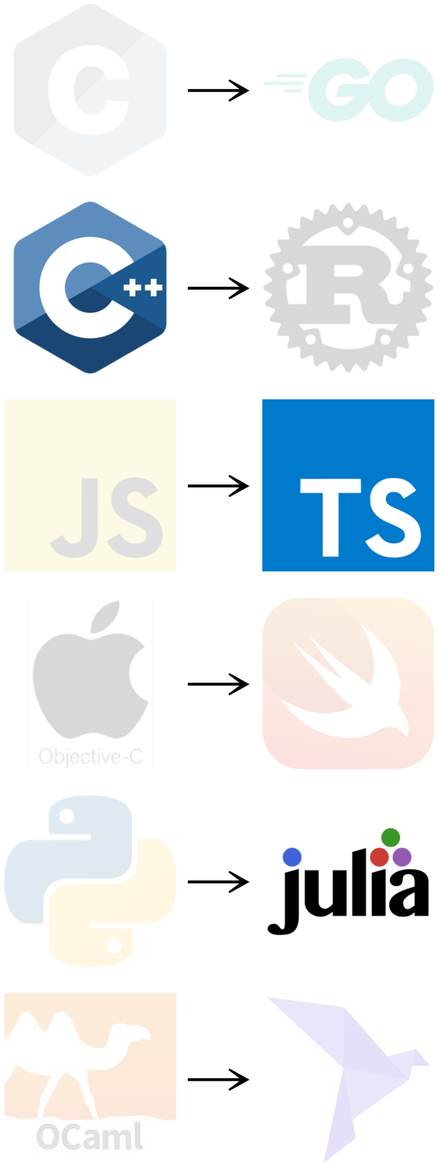


Type Inferred from Expressions

Local Type Inference

```
template <typename T>
auto gf(T a) { return a; }
auto inference() {
  auto sv = 42;              // infer from expression
  auto vv = vector{1, 2}; // infer from params
  auto gv = gf(42);          // infer from call
  return sv;                 // infer return type
}
```

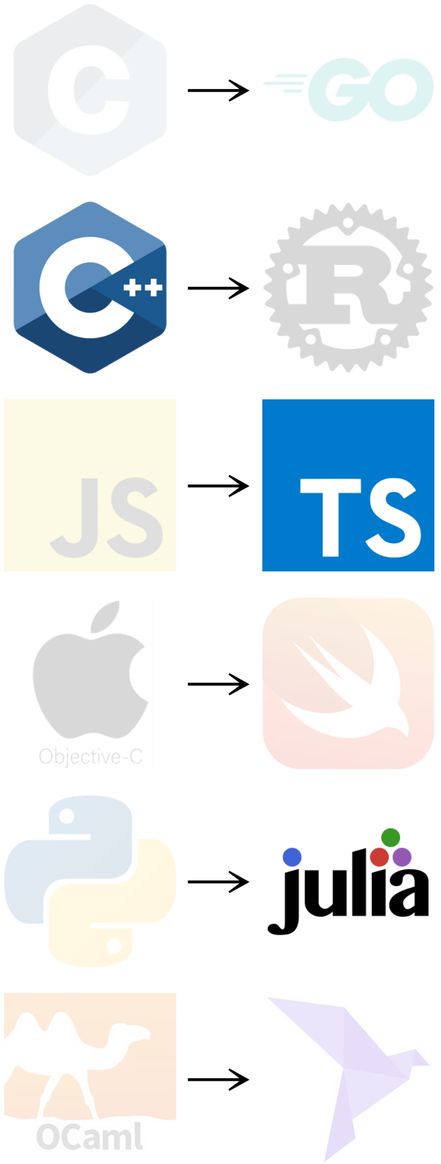
Type Inferred from Expressions



Local Type Inference

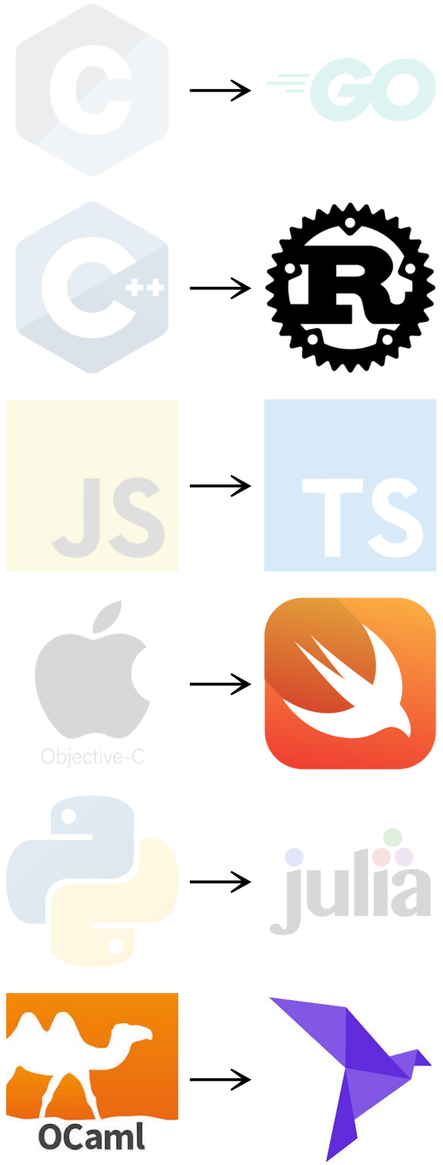
```
auto no_inference() {
  auto vv = vector{}; // cannot infer
  vv.push_back(42);   // ignored by error
}

function no_inference() {
  const fun : (x: any) => any = x => x+1; // too generic
  const fun2 : (x: number) => number = (x: number) => x+1;
}
```

Many Limitations



HM-Style Type Inference

```rust
fn inference() {
    let mut v: Vec<i32> = vec![];
    v.push(1);
    let f: impl Fn(i32) -> i32 = |x: i32| x + 1;
    f(1);
}
```



Lifts Some Limitations

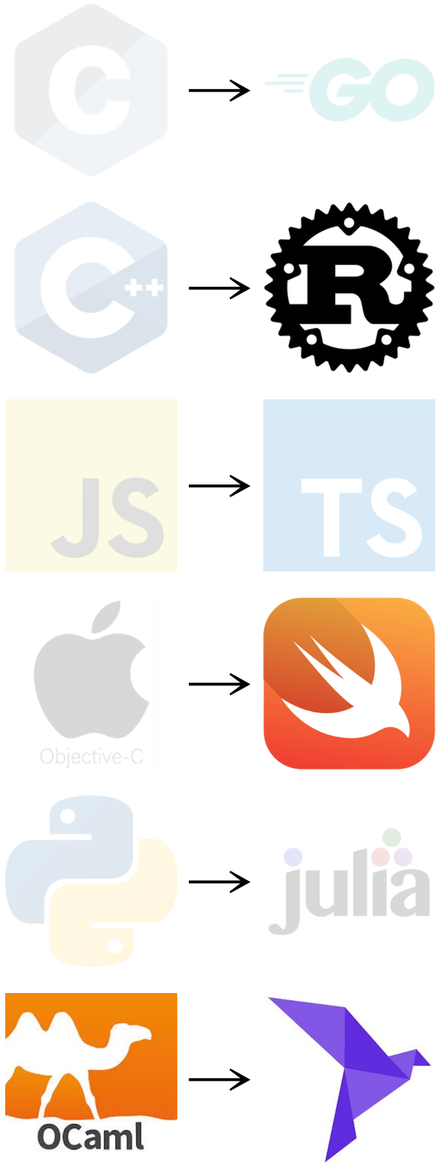
HM-Style Type Inference

Fully Decidable: No Annotations

Complex Error Messages

Function Annotations

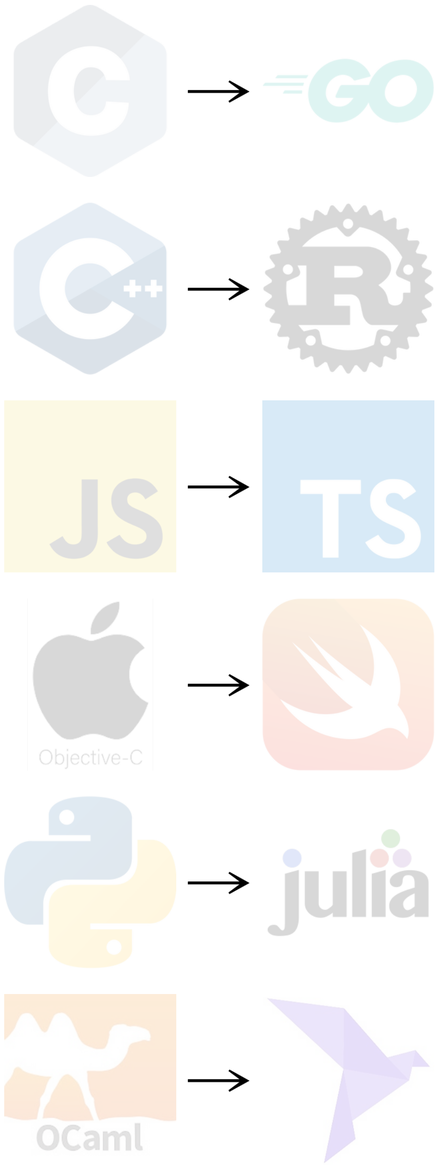
Limited, but Good Tradeoff



Type Inference

Prioritize Inference

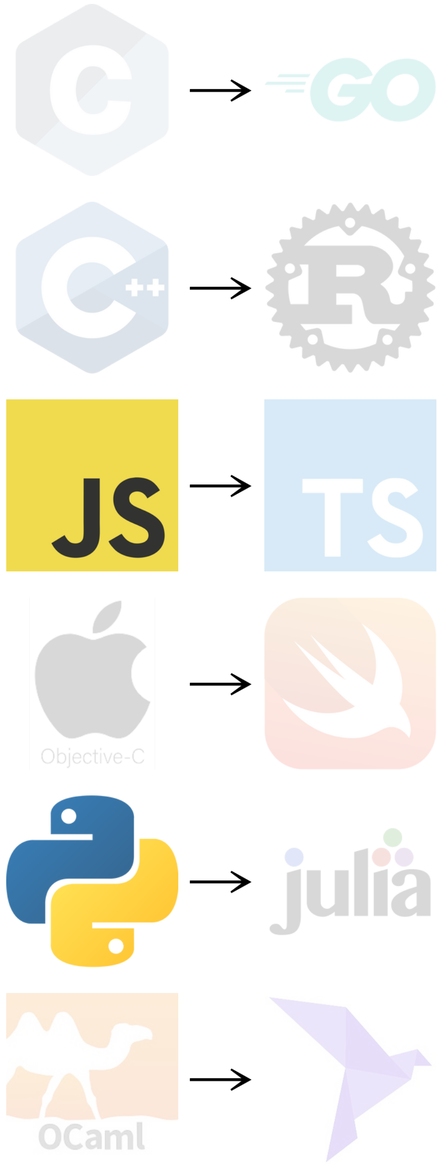
Guarding for Complexity



Algebraic Data Types

Dynamic Sum Types

```python
Vec2Like = tuple[float, float] | list[float]

def dot(v1: Vec2Like, v2: Vec2Like) -> float:
    return v1[0] * v2[0] + v1[1] * v2[1]
```



Represent Alternatives

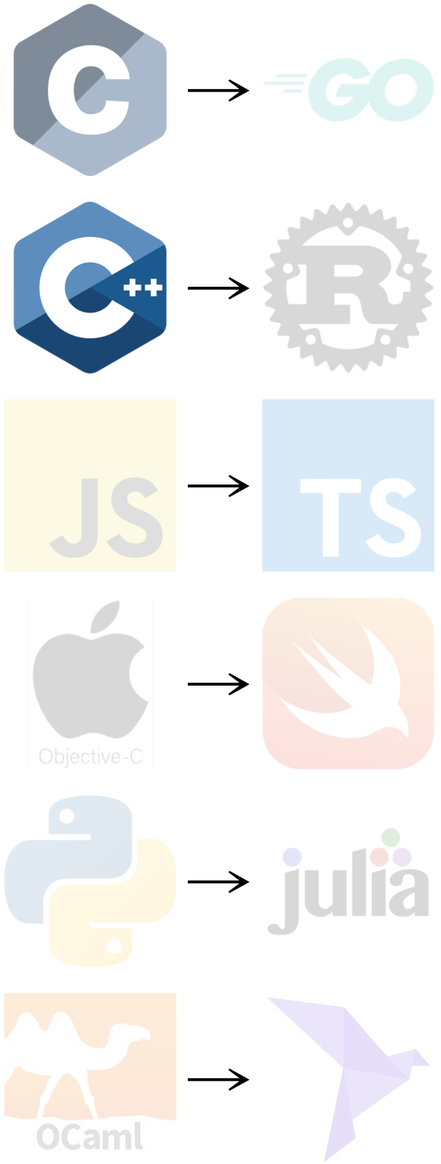
Runtime Errors on Use

Tagged Enums

```
enum Tag { Tag1, Tag2 };
struct TaggedUnion {
  Tag tag;
  union { Data1 d1; Data2 d2; };
};
```

Error Prone

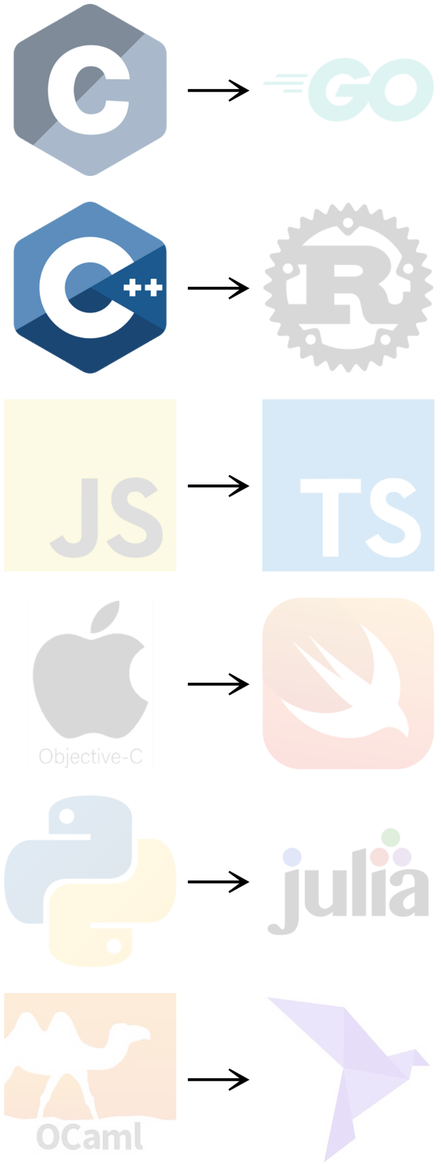
Not First Class



Tagged Enums

```
void dispatch(TaggedUnion u) {
    switch (u.tag) {
        case Tag1: use1(u.d1); break;
        case Tag2: use2(u.d2); break;
    }
}
```

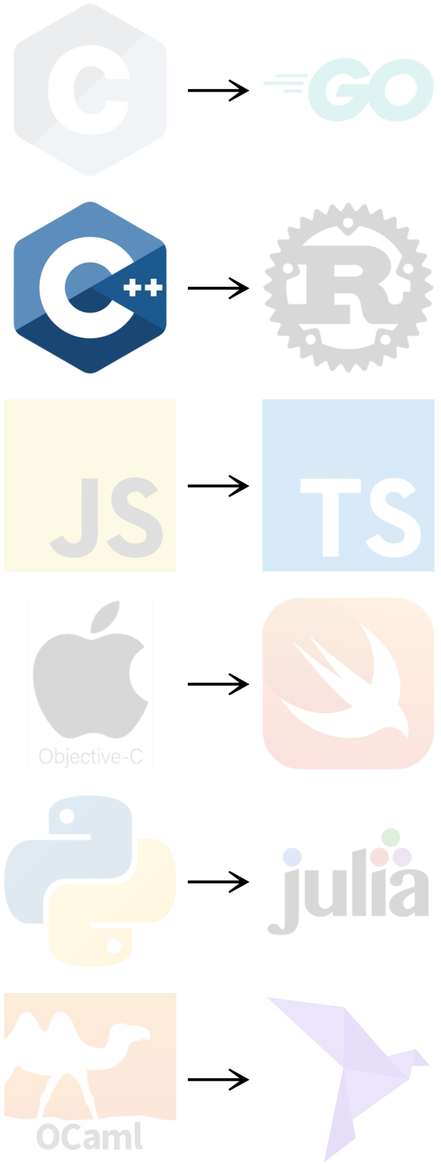
Manual Dispatch on Tag



Tagged Enums

```cpp
using Variant = variant<Data1, Data2>;
void call_use(Data1 d) { use1(d); }
void call_use(Data2 d) { use2(d); }
void dispatch(Variant v) {
  std::visit([](auto&& d) { call_use(d); }, v);
}
```

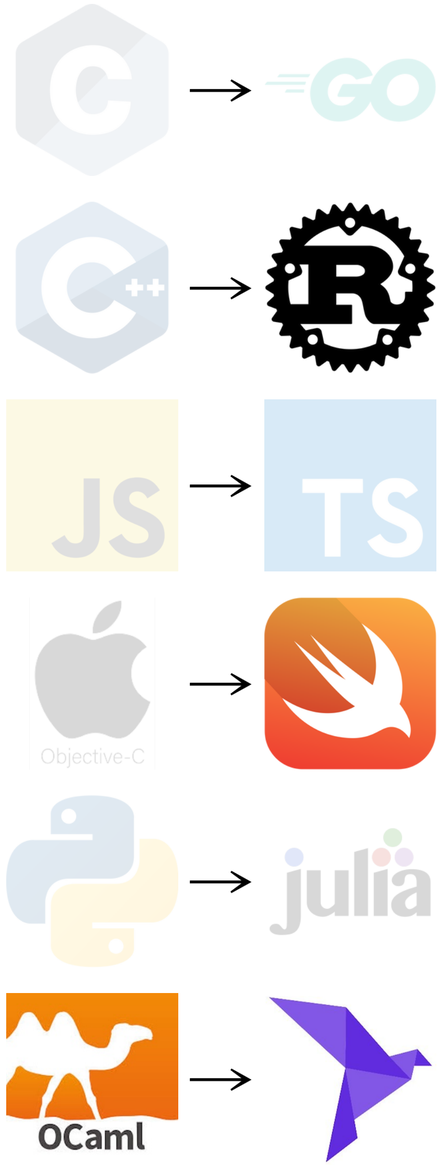
No comment



Sum Types

```
enum Variant {
    T1(Data1),
    T2(Data2),
}
```

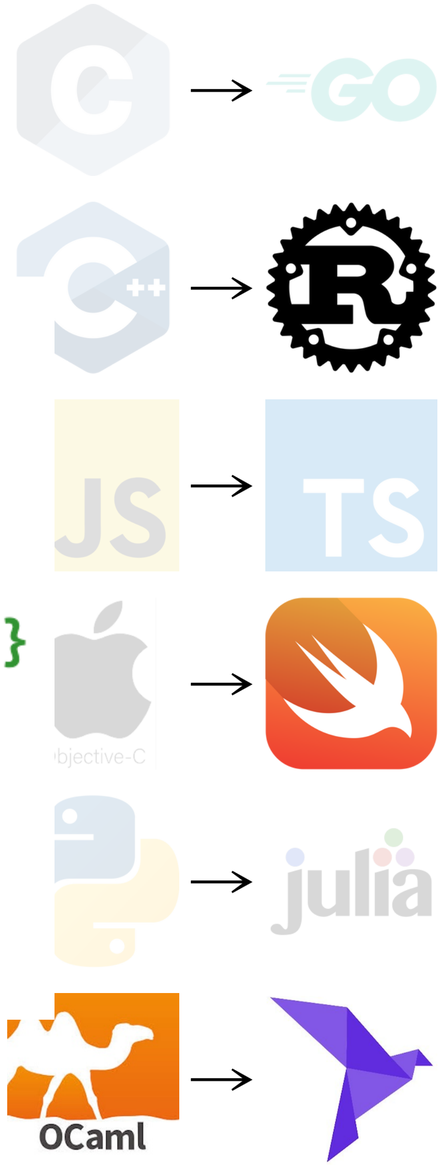
Variant with Payload



Sum Types

```rust
fn not_total(v: i32) -> Option<i32> {
    if v > 0 { Some(v) } else { None }
}
fn may_error(v: i32) -> Result<i32, String> {
    if v > 0 { Ok(v) } else { Err("error".to_string()) }
}
fn propagate_error(v: i32) -> Result<i32, String> {
    Ok(may_error(v)? + 1)
}
```

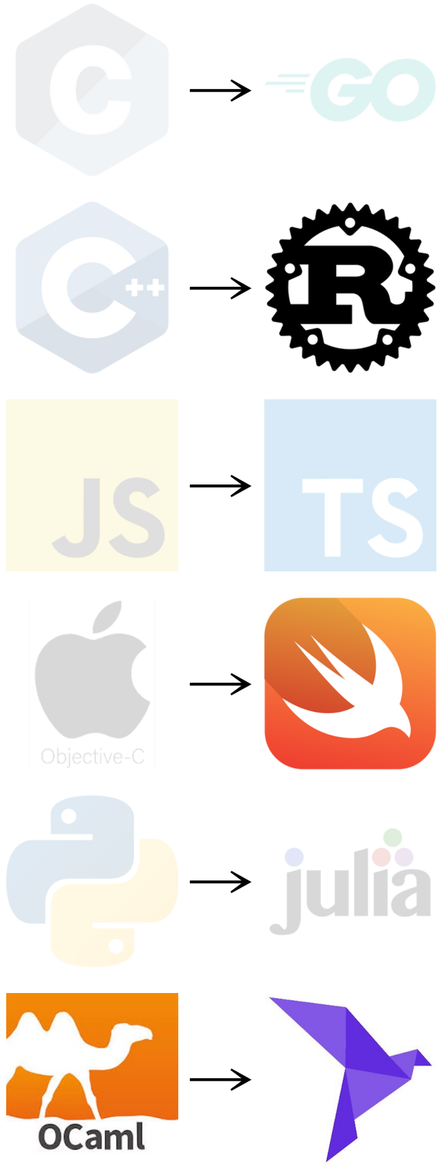
Used Extensively



Pattern Matching

```
fn pattern_match(v: Variant) {
    match v {
        Variant::T1(data: Data1) ⇒ use1(data),
        Variant::T2(data: Data2) ⇒ use2(data),
    }
}
```

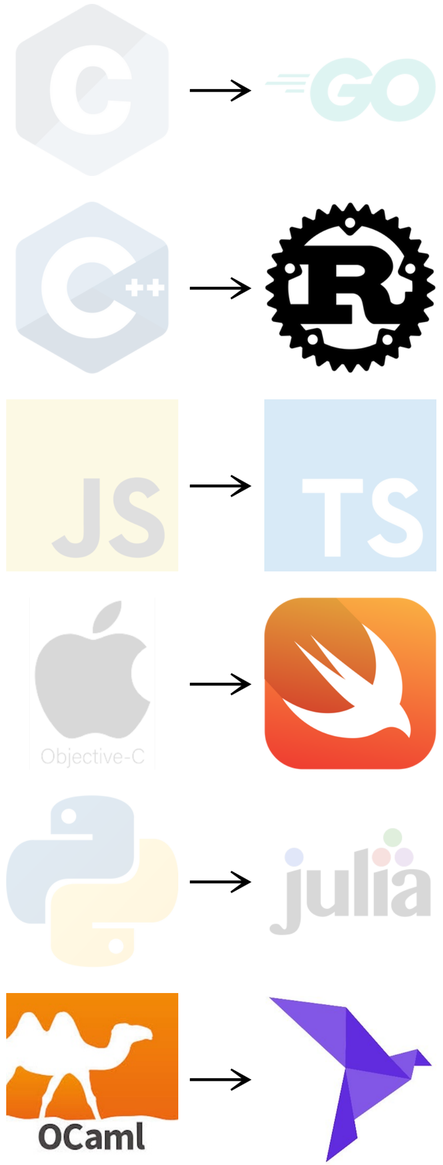
Dispatch via Pattern Matching



Pattern Matching

```
fn pattern_match(v: Variant) {
    match v {
        Variant::T1(data: Data1) ⇒ use1(data),
        Variant::T2(data: Data2) ⇒ use2(data),
    }
}
```

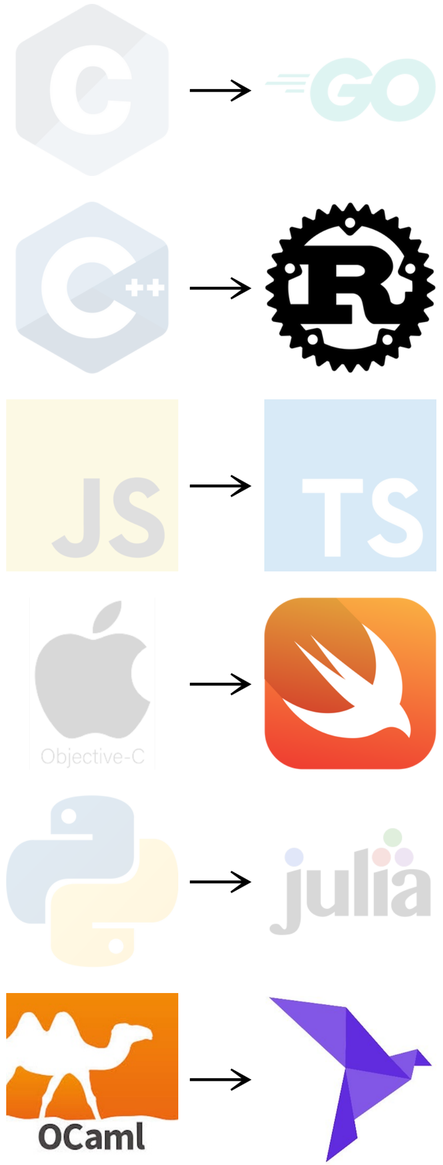
Dispatch via Pattern Matching



Pattern Matching

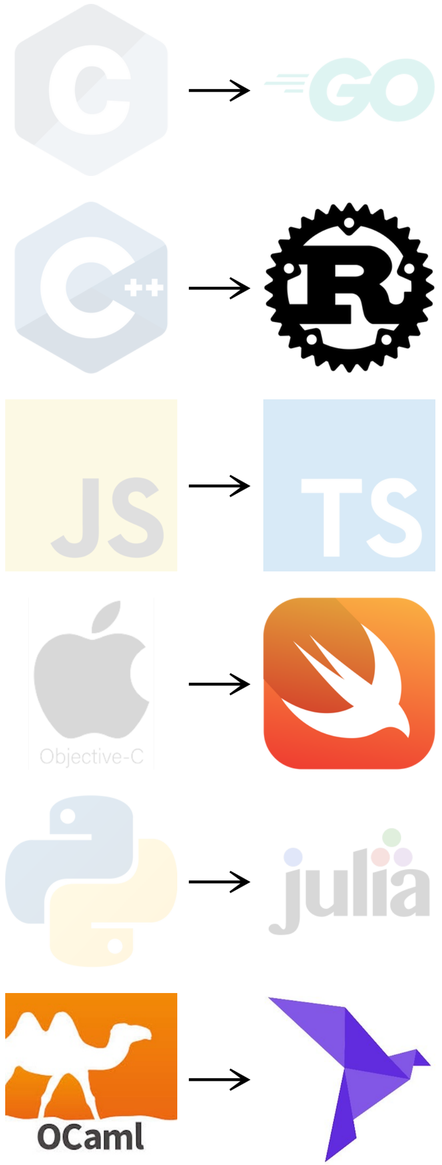
```
struct Point { x: i32, y: i32 }
fn match_point(p: Point) → i32 {
    match p {
        Point{ x: i32, y: 0 } ⇒ x,
        Point{ x: 0, y: i32 } ⇒ y,
        Point{ x: i32, y: i32 } ⇒ x + y,
    }
}
```

General Programming Tool



Pattern Matching

```rust
fn fibonacci(n: i32) -> i32 {
    match n {
        0 | 1 => 1,
        n: i32 if n > 0 => fibonacci(n - 1) + fibonacci(n - 2),
        _ => panic!("negative"),
    }
}
```



General Programming Tool

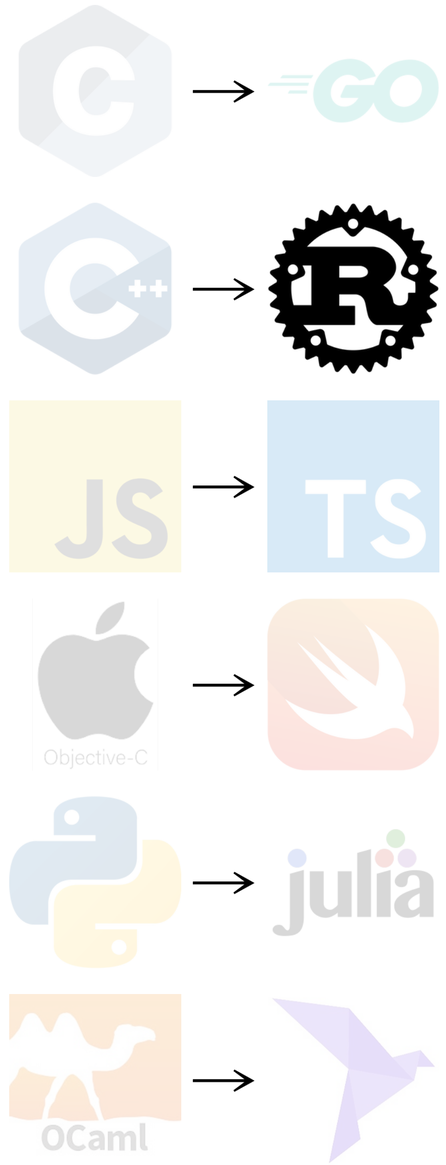
Lifetimes

Lifetime Annotations

Safe References via Lifetimes

Part of the Type System

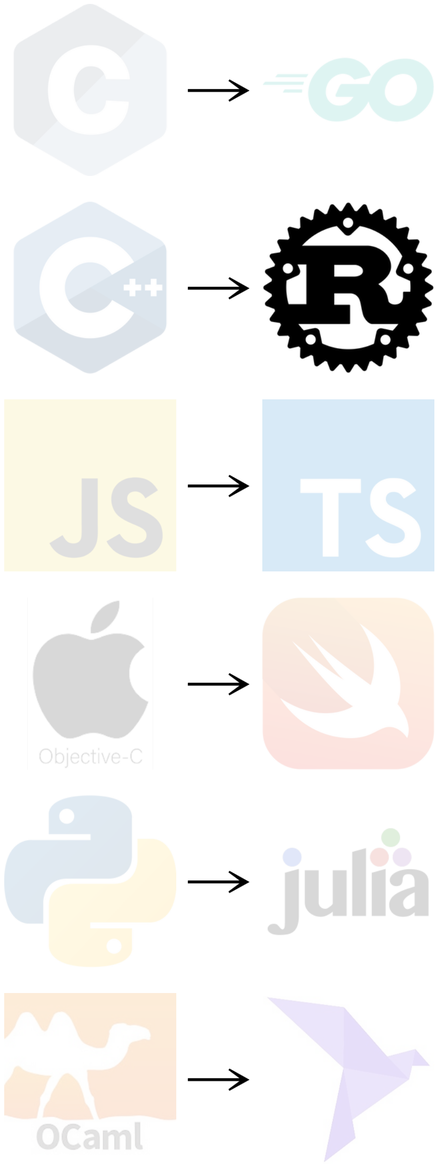
Mostly Automatic



Lifetime Annotations

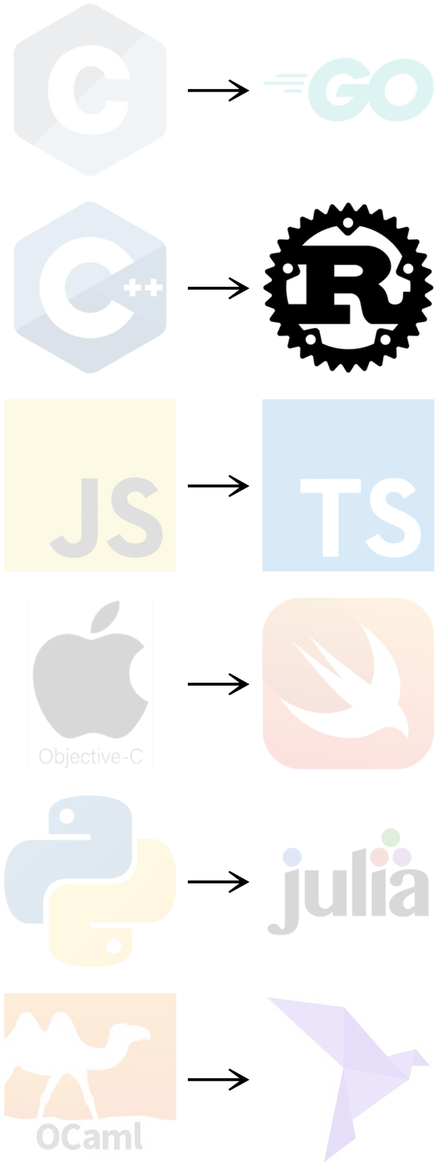
```rust
fn lifetimes() {
    let _sv: i32 = 42; // 'a
    let _sr: &i32 = &_sv; // 'a
    let _sc: &i32 = {
        // 'b
        let value: i32 = 42; // 'b
        &value // 'b
               // drop value → 'b invalid
    };
}
```

References to Valid Objects



Lifetime Annotations

```rust
struct Ref<'a, T: 'a> {
    x: &'a T,
}

fn identity<'x, T>(x: &'x T) -> &'x T { x }
```

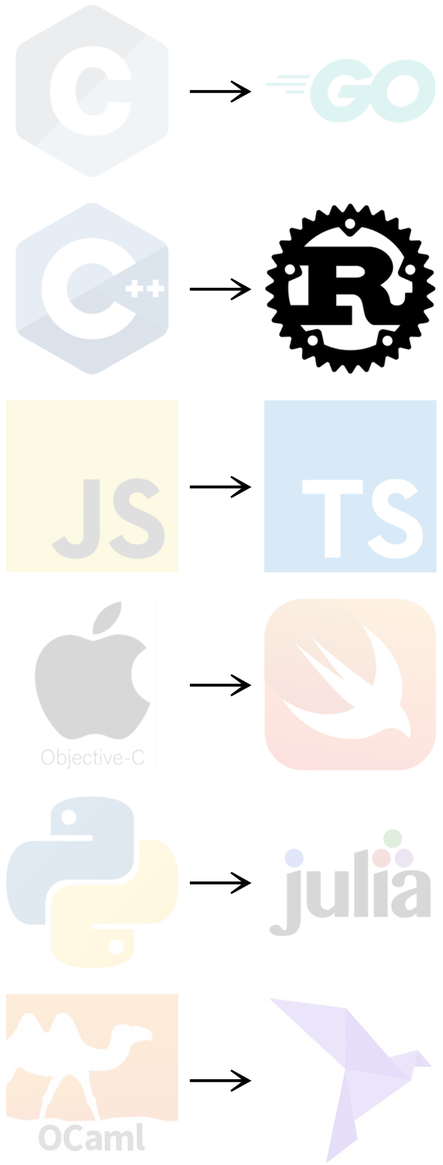


Mostly Inferred, Some Explicit

Lifetime Annotations

Inference Simplifies Lifetimes

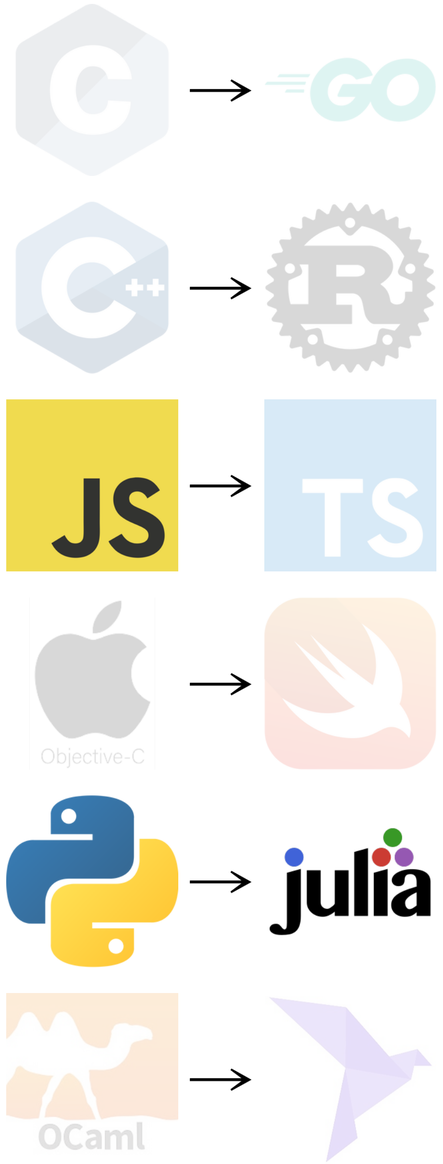
Generics Simplifies Lifetimes



Generics

Dynamic Polymorphism

```python
def dynamic(x: int | str | list[int]) -> int:
    if isinstance(x, str):
        return 0
    elif isinstance(x, list):
        return sum(x)
    else:
        return x + 1
```



Free-for-all Polymorphism

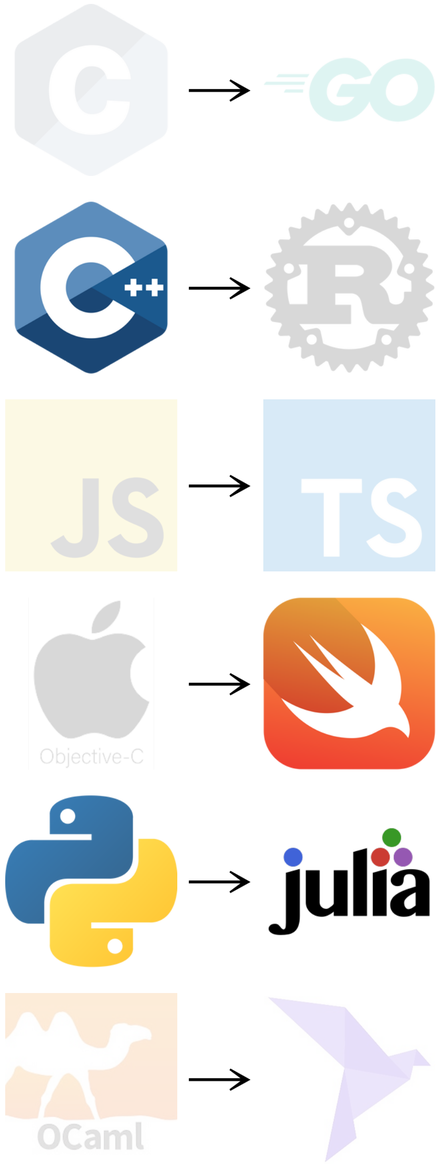
SubType Polymorphism

```
struct Base                { int f() { return 42; } };
struct Derived : Base { int f() { return 43; } };
auto call_f(Base* b) { return b->f(); }
```

Object-Oriented Programming

Many Drawbacks

Julia's Multiple Dispatch



Compile-Time Polymorphism

```
template<typename T>
void parametric(T a) { use(a); }

void ad_hoc(Data1 a) { use1(a); }
void ad_hoc(Data2 a) { use2(a); }
```

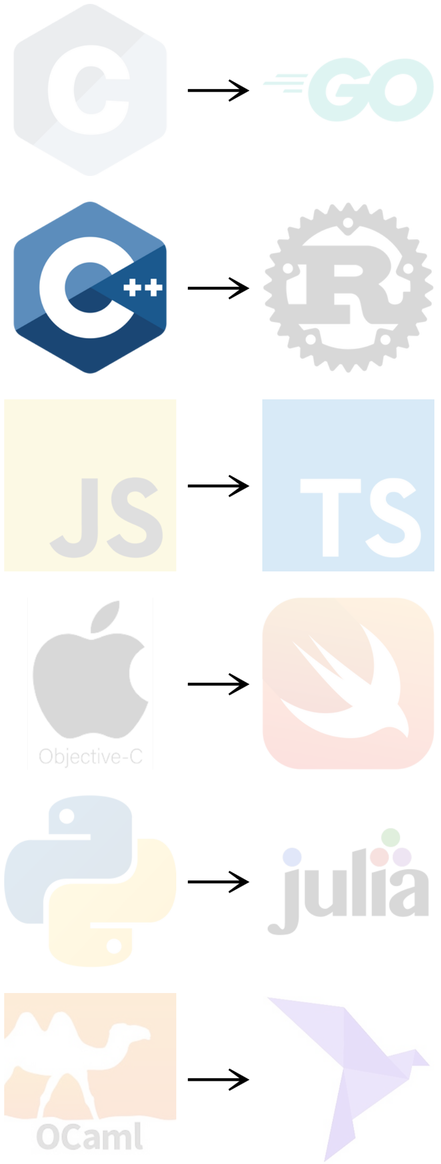
Ad-hoc or Parametric

Modern Generic Libraries

Unconstrained Generics

```
template<typename T>
auto gsum(T a, T b) { return a + b; }

void sum() {
  auto s1 = gsum(1, 2);       // ok
  auto s2 = gsum(1, 3.0);     // type mismatch
  auto s3 = gsum(vector{1},   // no operator+
                 vector{2}); // instantiation error
}
```

Errors during Instantiation

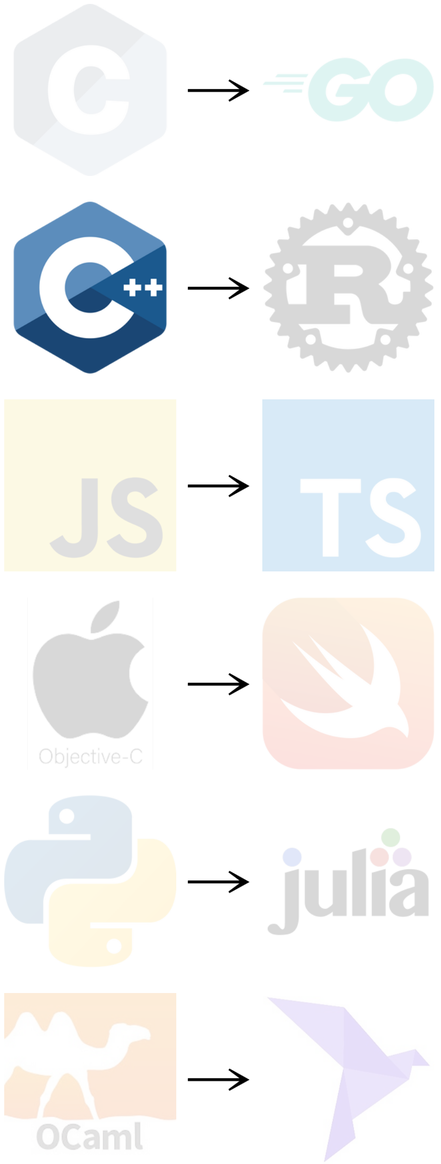


Constrained Generics

```
template<typename T> concept Equatable =
  requires (T a, T b) { a == b; };
template<Equatable T>
auto eq(T a, T b) { return a == b; }
template<Equatable T>
auto prod(T a, T b) { return a * b; }
```

Concepts Check Instantiation

No Check on Implementation

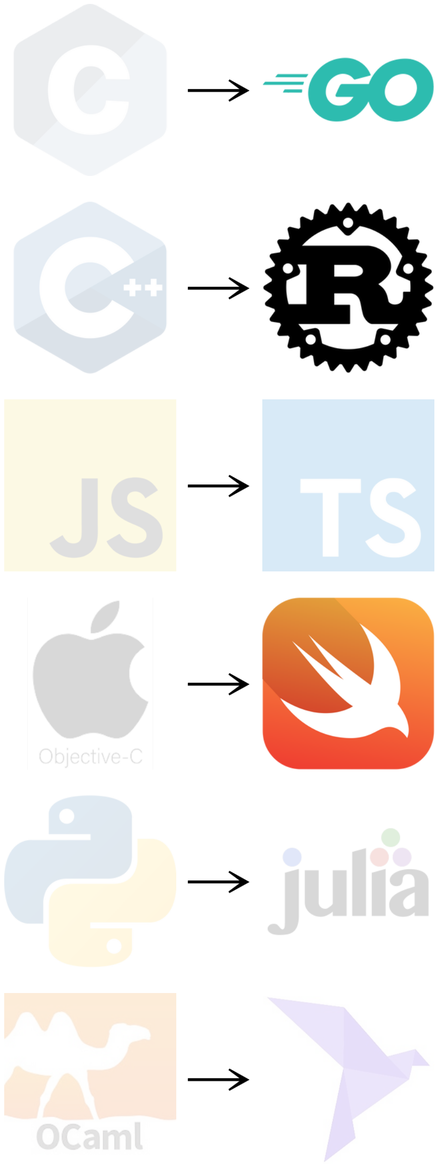


Traits

```
fn eq<T: PartialEq>(x: T, y: T) → bool {
    x == y
}

fn eq_err<T>(x: T, y: T) → bool {
    x == y
}
```

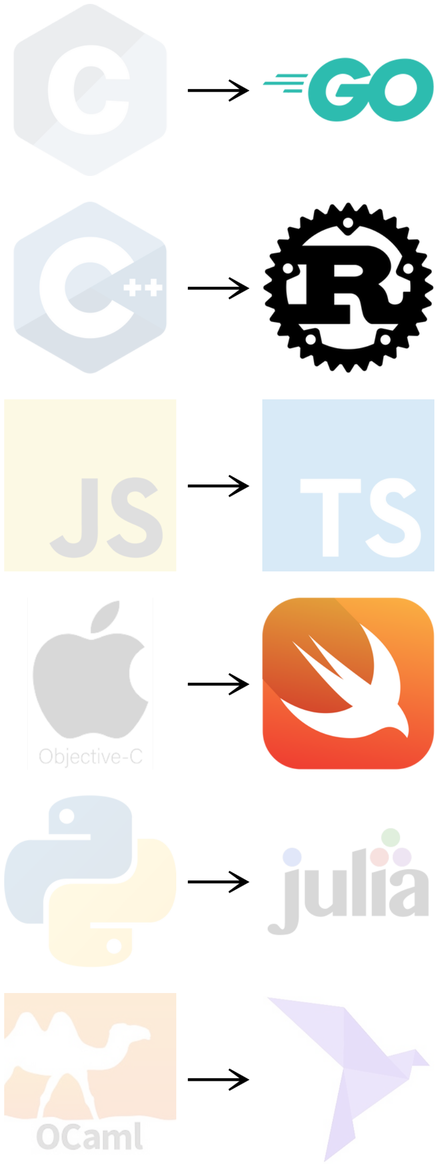
Check Instantiation and Impl.

Used Throughout the Language



Traits

```rust
trait IsNil {
    fn is_nil<'self>(&'self self) -> bool;
}

impl IsNil for i32 {
    fn is_nil<'self>(&'self self) -> bool {
        *self == 0
    }
}

impl IsNil for Vec<i32> {
    fn is_nil<'self>(&'self self) -> bool {
        self.is_empty()
    }
}
```



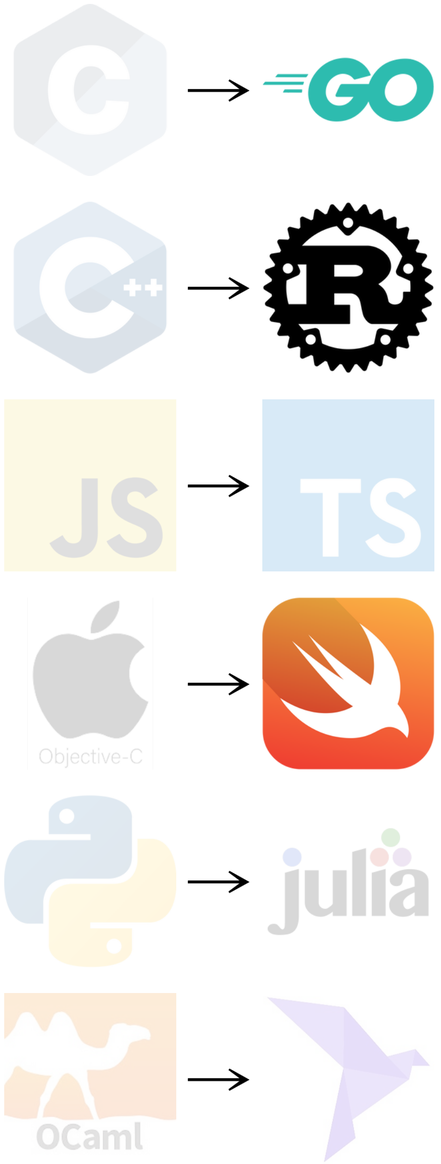
Traits

Traits define Interfaces

Constrain Generics

Ad-hoc Implementations

Operator Overloading

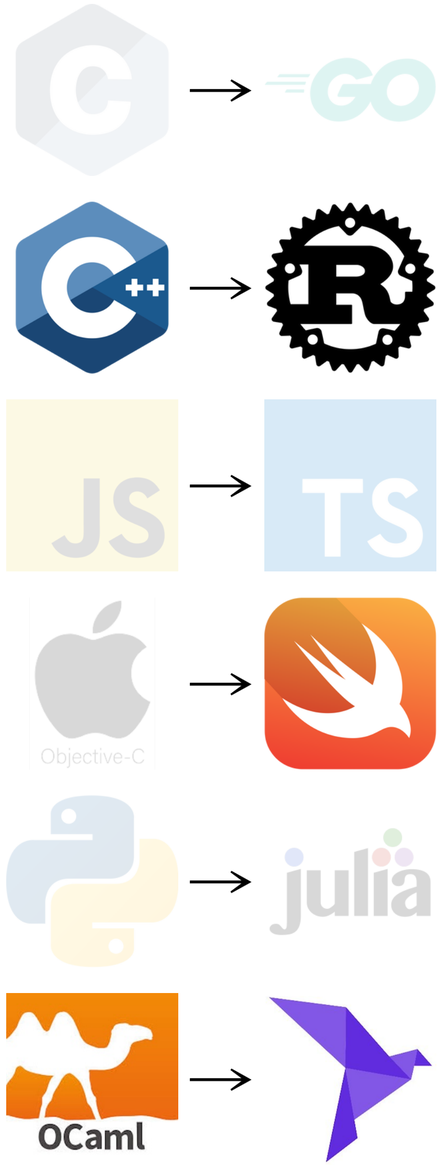


Generic Compilation

Monomorphization

Dynamic Dispatch

Multiple Dispatch



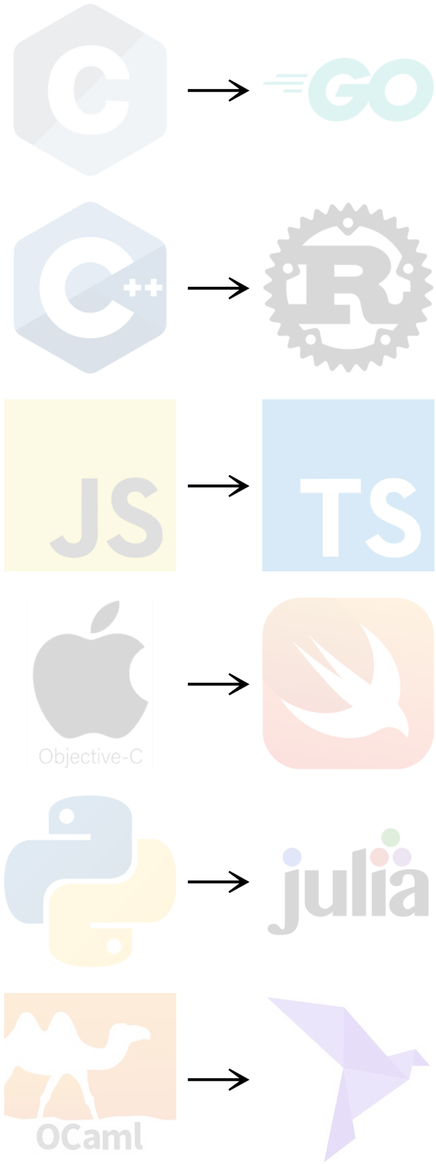
Generics

Compile-Time

Checked aka Traits

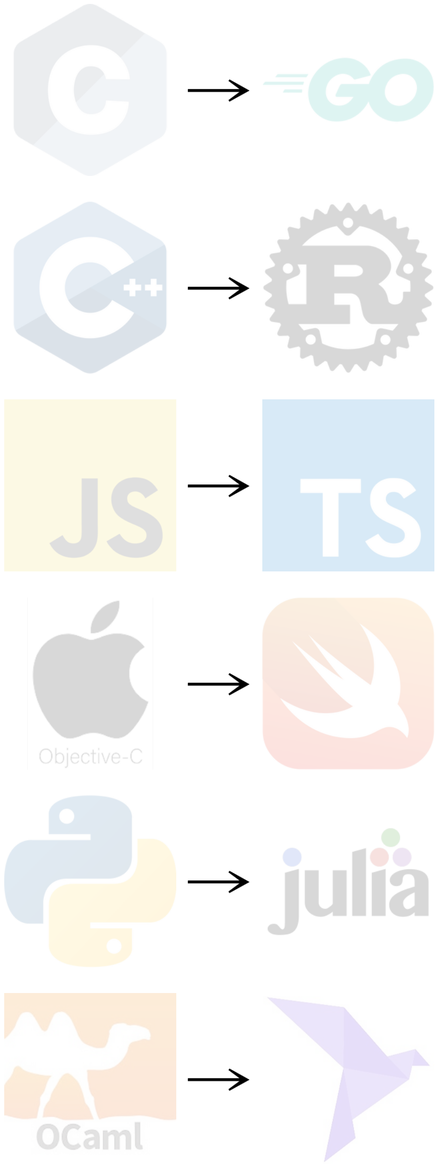
Monomorphized

Base of Language and StdLib

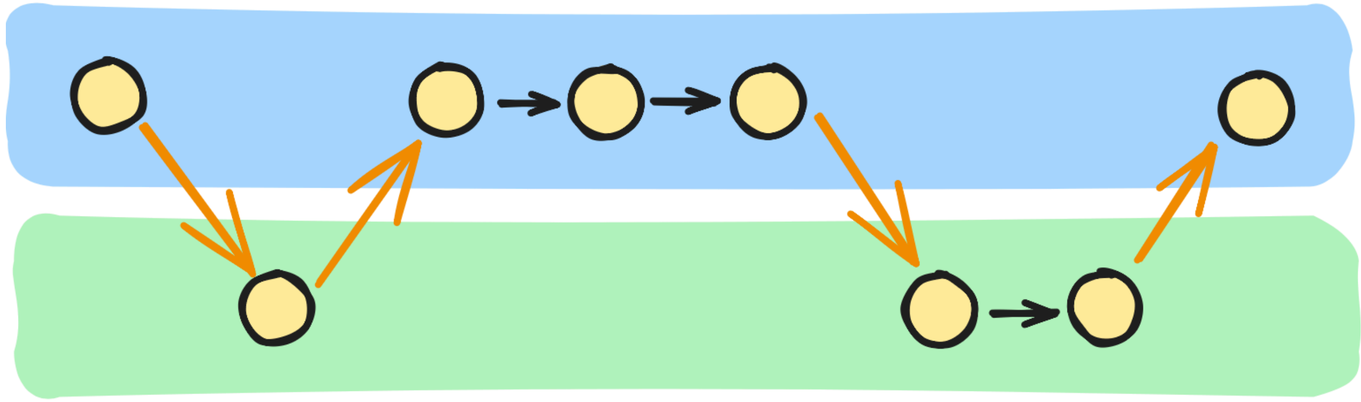


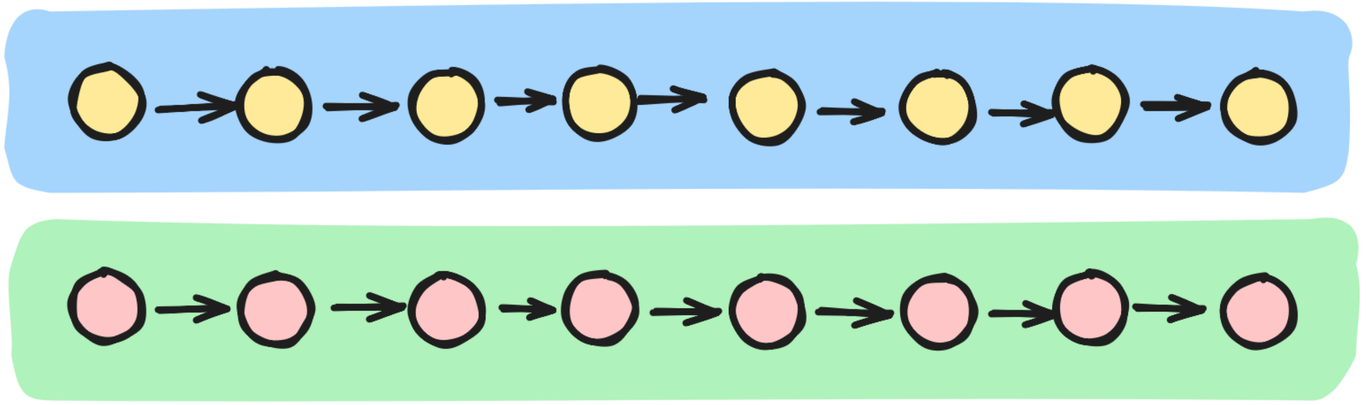
Concurrency

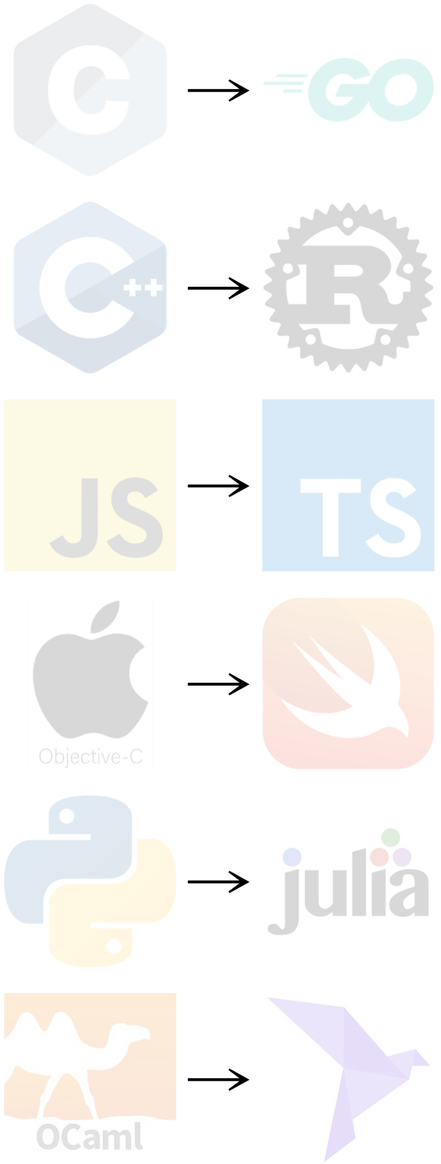
Concurrency is not Parallelism



Concurrency-vs-Parallelism



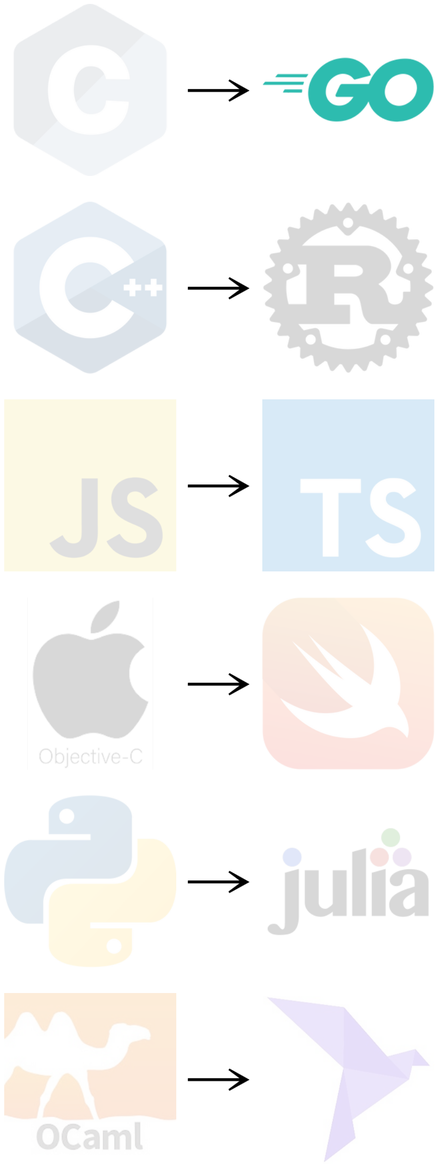




Concurrency

```go
func main() {
    channel := make(chan int)
    go concurrent(channel)
    value := <-channel
    println(value)
}

func concurrent(channel chan int) {
    channel <- 42
}
```

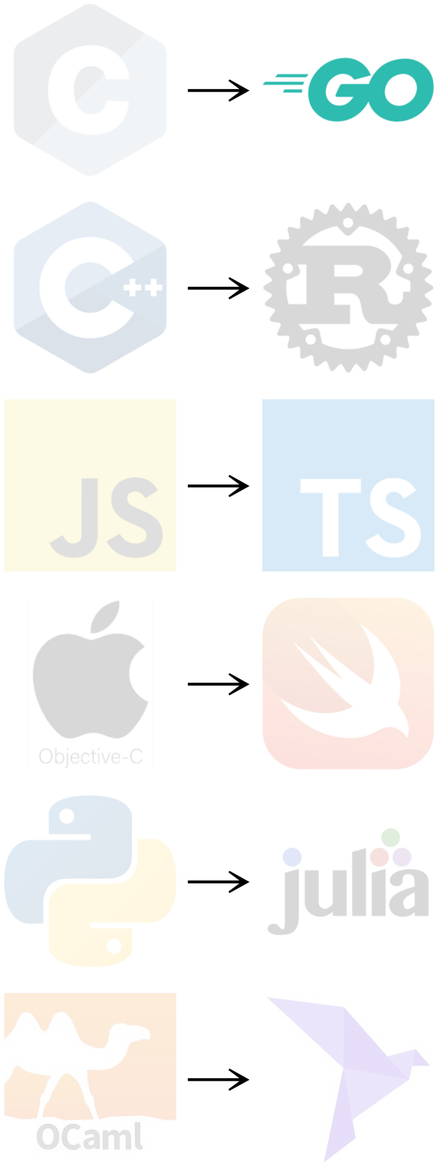
Language Support



Concurrency

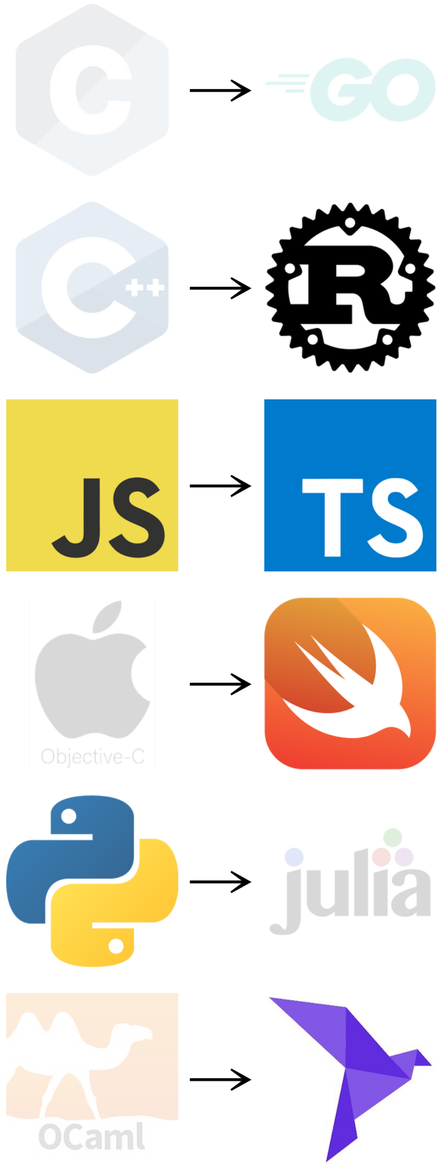
```go
func main() {
    channel := make(chan int)
    go concurrent(channel)
    value := <-channel
    println(value)
}

func concurrent(channel chan int) {
    channel <- 42
}
```

Language Support



Concurrency

```rust
#[tokio::main]
▶ Run | Debug
async fn main() {
    let value: i32 = concurrent().await;
    println!("{}", value);
}

async fn concurrent() -> i32 {
    42
}
```



Language Support

Parallelism

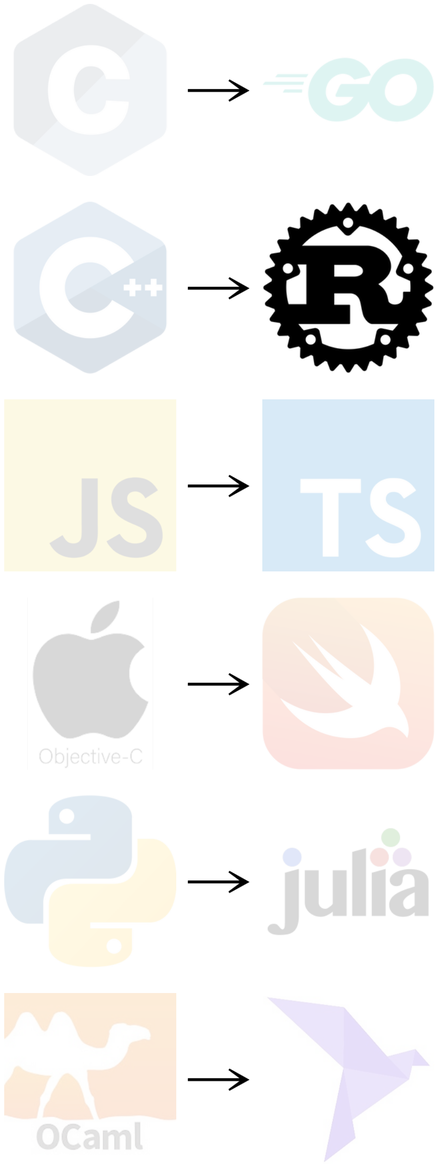
```rust
pub fn parallel() {
    let v = Vec::from([1; 1000]);
    let _ssum = v.iter().map(|x| x * 2).sum::<i32>();
    let _psum = v.par_iter().map(|x| x * 2).sum::<i32>();
}
```

```cpp
int parallel() {
  auto v = std::vector<int>(1000, 1);
  tbb::parallel_for(0, 1000, [&](int i) { v[i] *= 2; });
}
```

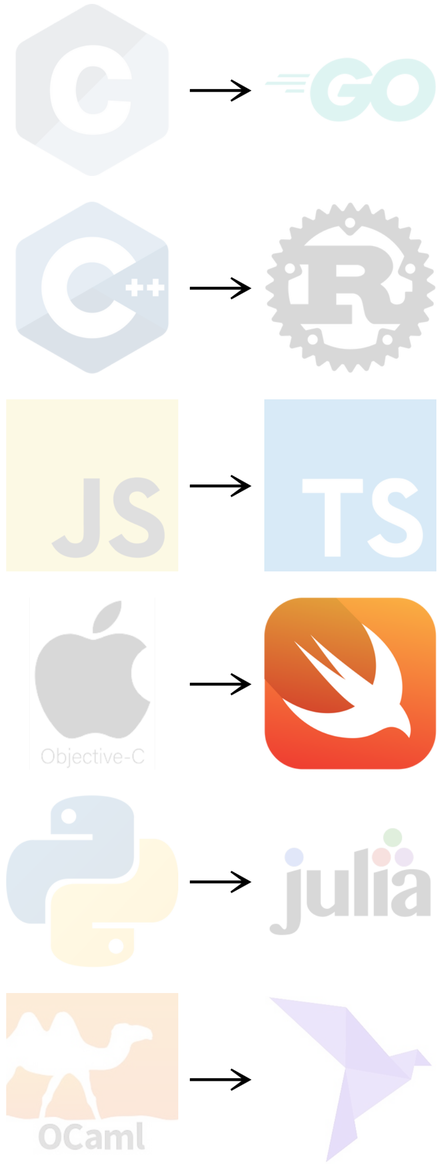
Library Support

Data-Race Safety

```rust
pub fn data_race() {
    let mut v = Vec::from([1; 1000]);
    v.par_iter()
        .map(|x| {
            v[0] = 0;
            x * 2
        })
        .sum::<i32>();
}
```



Data-Race Safety



A Flexible Type System for Fearless Concurrency

Mae Milano
University of California, Berkeley
Berkeley, CA, USA
mpmilano@berkeley.edu

Joshua Turcotti
University of California, Berkeley
Berkeley, CA, USA
jturcotti@berkeley.edu

Andrew C. Myers
Cornell University
Ithaca, NY, USA
andru@cs.cornell.edu

Abstract

This paper proposes a new type system for concurrent programs, allowing threads to exchange complex object graphs without risking destructive data races. While this goal is shared by a rich history of past work, existing solutions either rely on strictly enforced heap invariants that prohibit natural programming patterns or demand pervasive annotations even for simple programming tasks. As a result, past systems cannot express intuitively simple code without unnatural rewrites or substantial annotation burdens. Our work avoids these pitfalls through a novel type system that provides sound reasoning about separation in the heap while remaining flexible enough to support a wide range of desirable heap manipulations. This new sweet spot is attained by enforcing a heap domination invariant similarly to prior work, but tempering it by allowing complex exceptions that

1 Introduction

The promise of a language with lightweight, safe concurrency has long been attractive. Such a language would statically ensure freedom from destructive races, avoiding the cost of synchronization except when concurrent threads explicitly communicate. Our goal is to obtain this "fearless concurrency" [35] for a language with pervasive mutability at its core. Broadly speaking, past efforts to design such a language fall into three camps. Some, like Rust [36], simplify reasoning by severely limiting the shape of representable data structures—making the implementation of common data structures, like the doubly linked list, unapproachable by non-experts[1]. In others [17, 26, 28, 29, 33, 46], harsh limitations on aliasing cause data structure traversal and manipulation to involve significant mutation of the object graph even for simple computations—for example, in these systems remov-

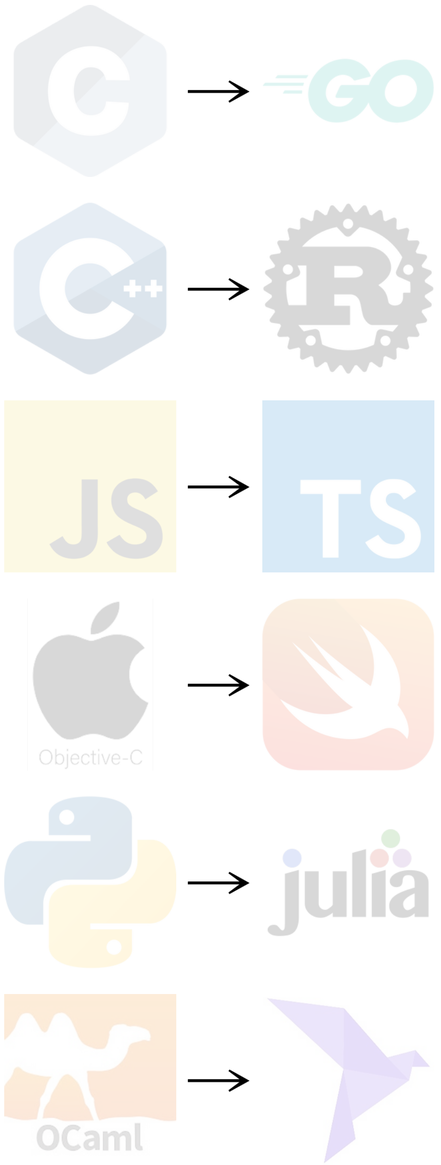
Concurrency and Parallelism

Concurrency in Language

Parallelism in Libraries

Some Safety Guarantees

Still not Ubiquitous



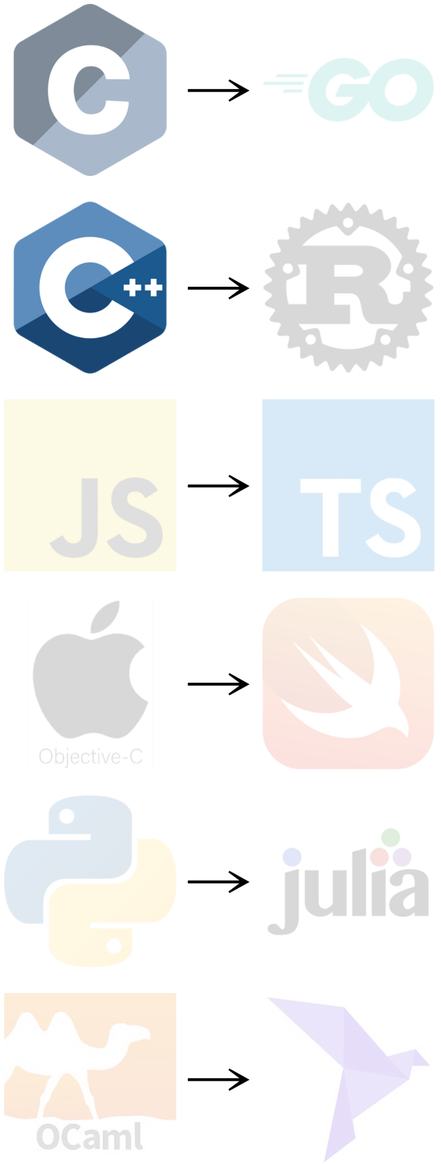
Metaprogramming

Code Generation

```c
#define SUM(a, b) (a + b)
int use_macro() { return SUM(1, 2) * 3; }

constexpr int csum(int a, int b) { return a + b; }
int use_constexpr() { return csum(1, 2) * 3; }
```

C's Macros

C++'s/Zig's Constexpr

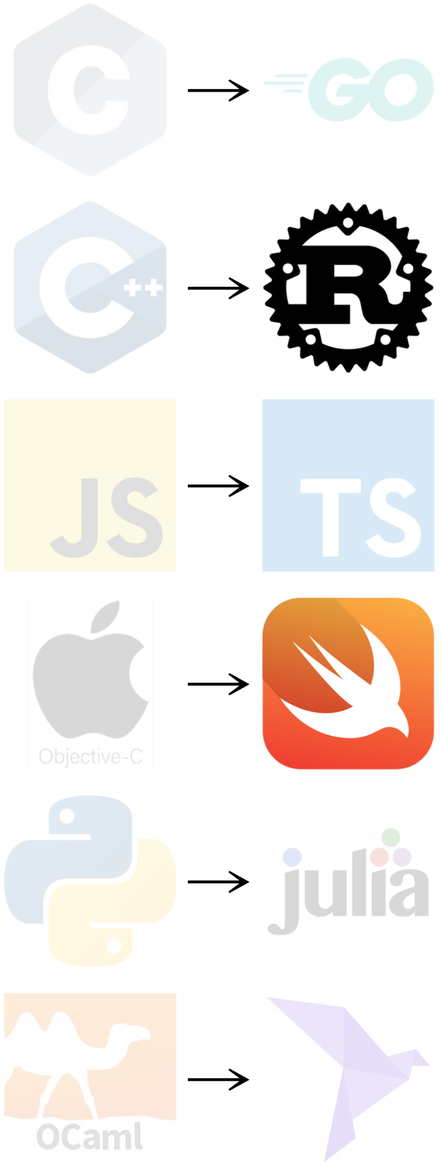


Code Generation

```rust
#[derive(Parser)]
struct CliArgs {
    name: String,
    #[arg(short, long)]
    help: Option<bool>,
}
```

Rust Hygienic Macros

Swift Compiler Plugins



Contemporary Desiderata

Desiderata for Contemporary Languages

Simpler References and Types

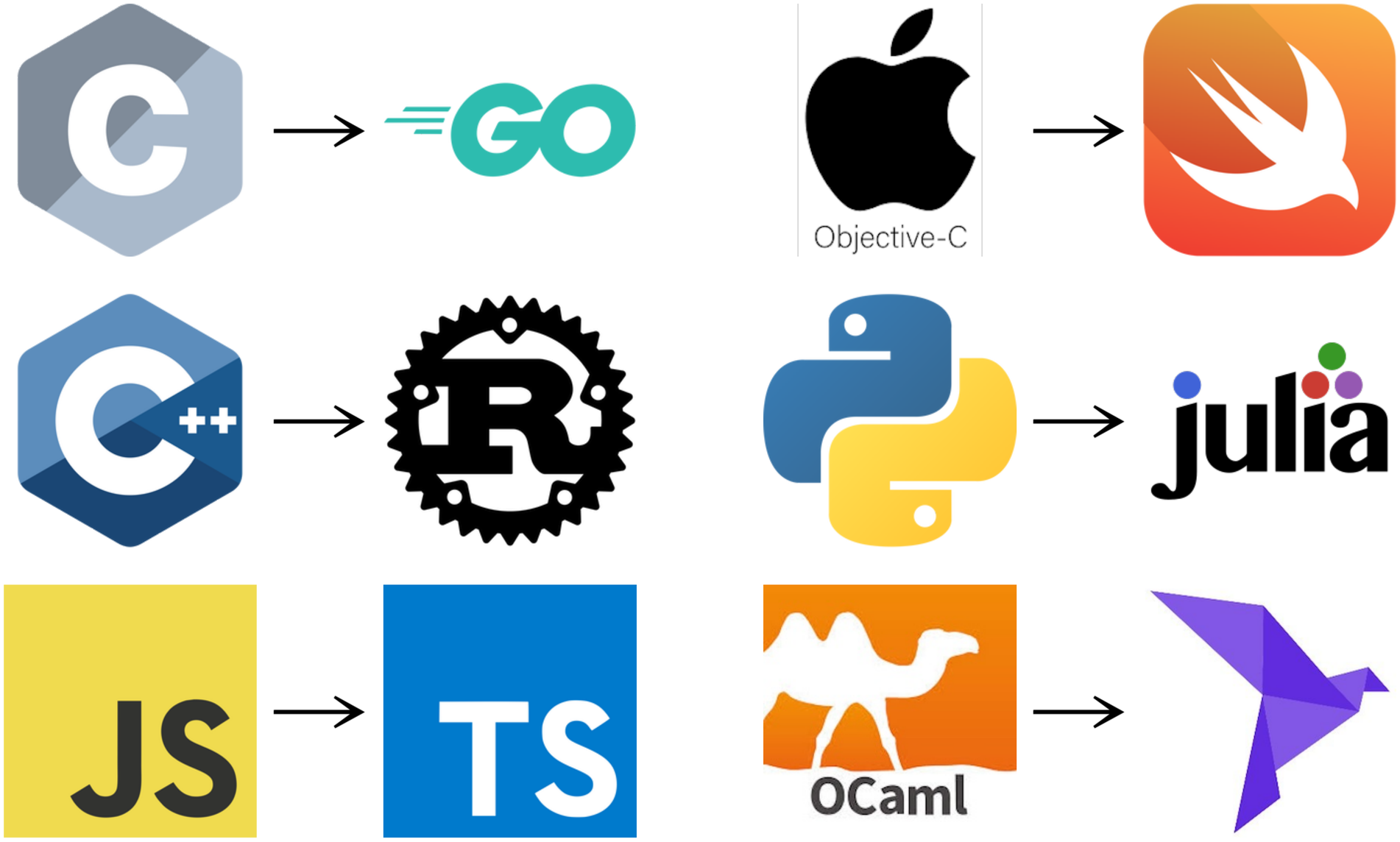
Target Parallel CPU and GPU

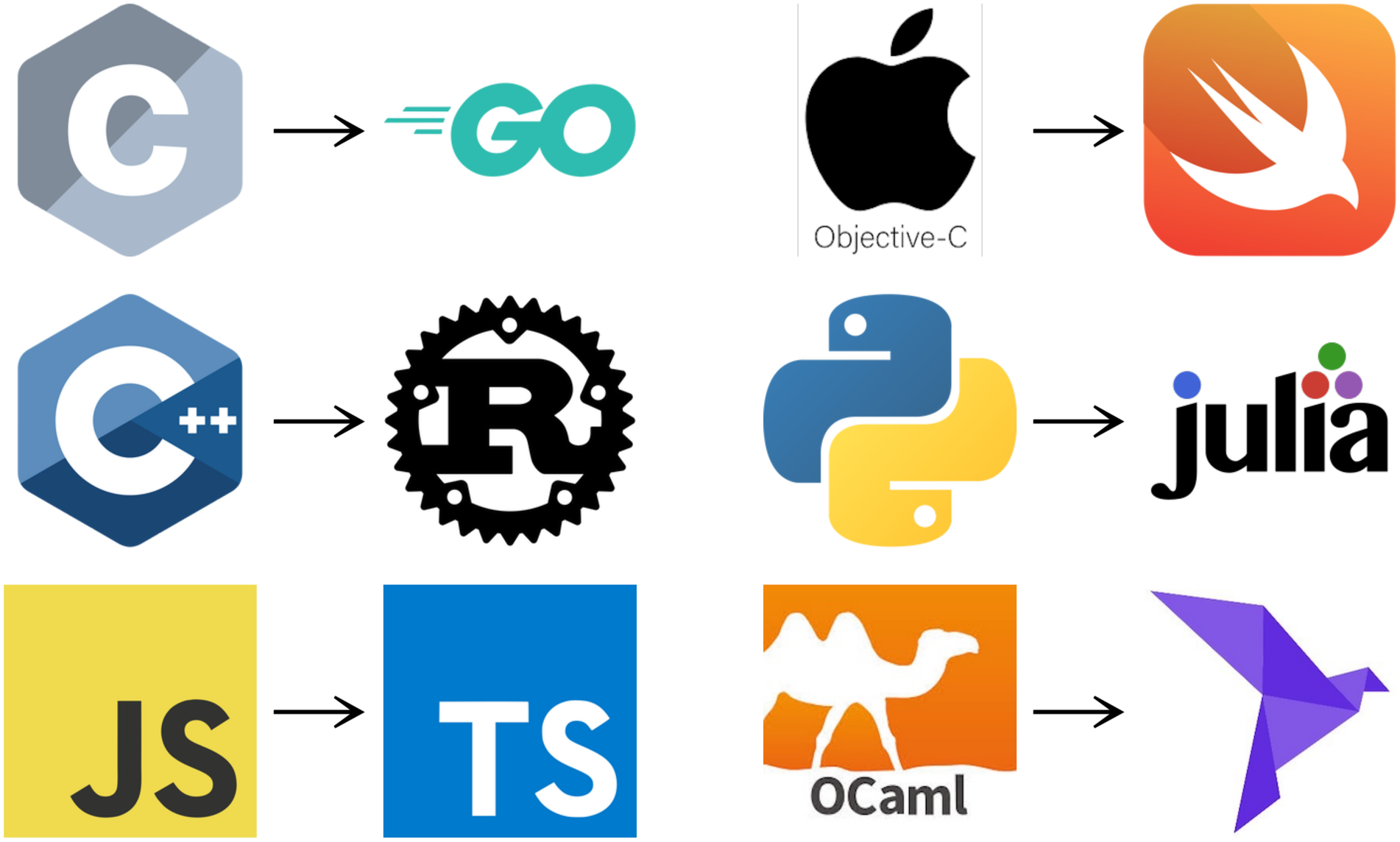
Heterogenous Systems

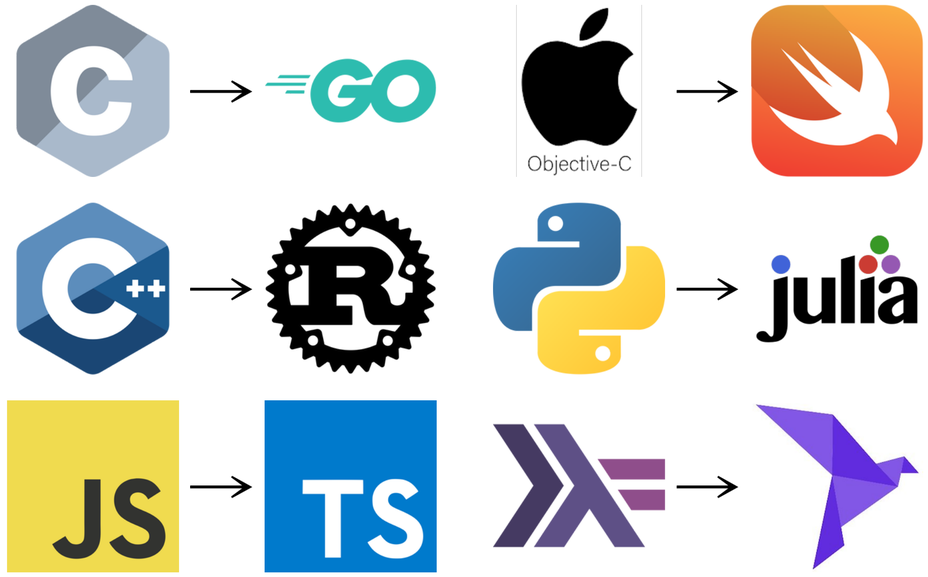
Distributed Systems

Modern Languages Rule!

Fabio Pellacini, FIM, UniMoRe

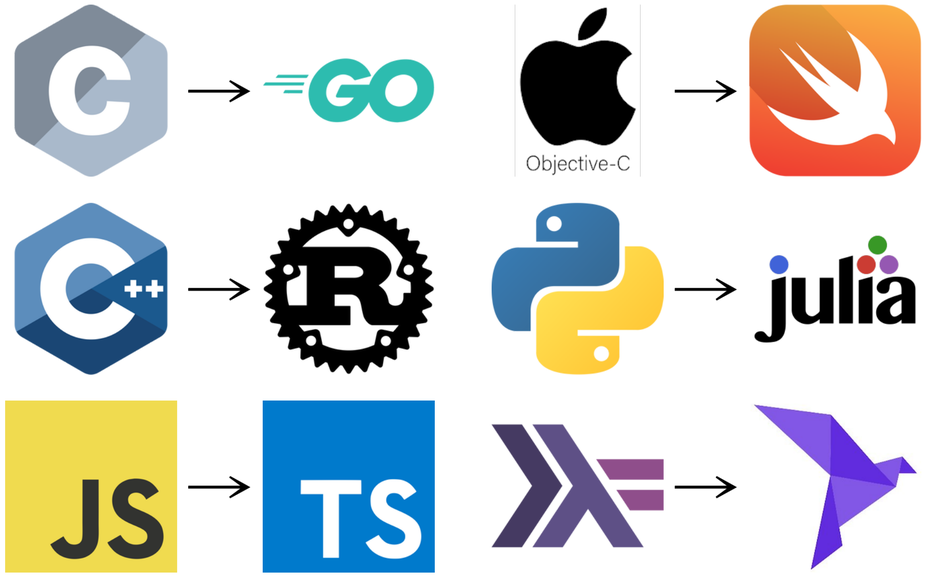

C
GO
Objective-C
C++
R
julia
JS
TS
OCaml

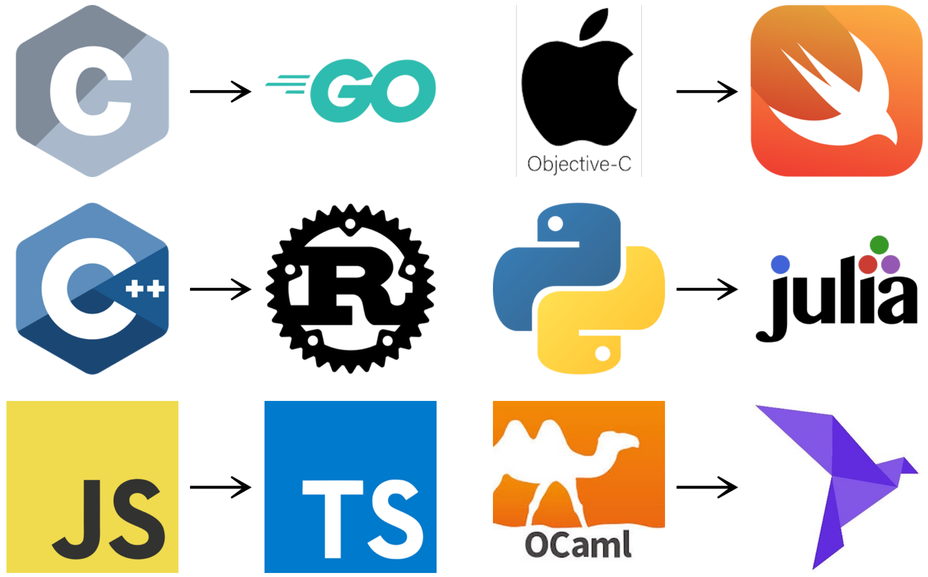

C
GO
Objective-C
C++
R
julia
JS
TS
OCaml

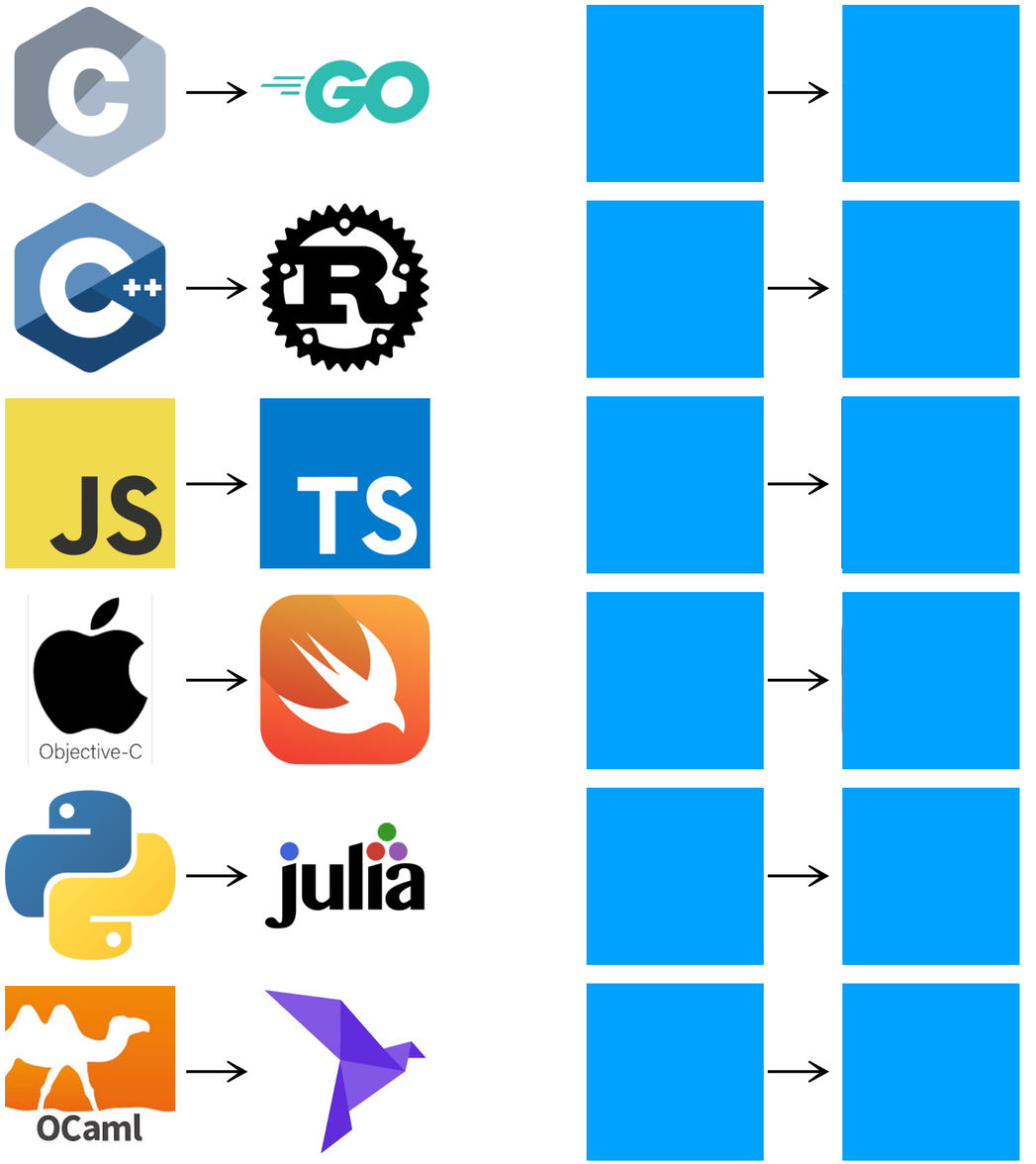

Objective-C
julia
JS
TS


Objective-C
julia
JS
TS


Objective-C
julia
JS
TS


Objective-C
julia
JS
TS


Objective-C
julia
JS
TS
OCaml


Objective-C
julia
JS
TS
OCaml