CO523 – Programming Languages
Assignment 02: Type System Study

E/20/449

WIJEWARDHANA A.R.S.S.

Task 1: Data Types and Type Systems

(a) Data Types

Data types are classifications that tell the compiler or interpreter how the programmer intends to use the data. They determine what values a variable can take and what operations can be performed on them.

1. Primitive Data Types

These are the most basic types provided by a programming language as building blocks. They usually have a direct mapping to hardware memory representation.

- **Java:** Includes int, boolean, char, and double. These are predefined by the language.

Java

```
int age = 25;           // 32-bit integer
boolean isStudent = true; // logical value
```

- **Python:** Includes int, float, bool, and str. Unlike Java, these are objects, but they serve as the fundamental types.

Python

```
price = 19.99  # float
name = "Gemini" # string
```

2. Composite Data Types

Also known as compound types, these are types derived from one or more primitive types. They allow multiple values to be grouped together.

- **C:** The most common composite type is the **Array**.

C

```
int scores[5] = {90, 85, 70, 95, 80}; // A collection of 5 integers
```

- **Python:** Uses **Lists** and **Dictionaries**.

Python

```
user_data = ["Alice", 30, "Engineer"] # List containing string and int
```

3. User-defined Data Types

These allow programmers to define their own types to model real-world entities. They are often built using primitives and composite types.

- **C:** Uses struct to group different types under one name.

C

```
struct Student {
   char name[50];
   int id;
   float gpa;
};
```

- **Java:** Uses **Classes** to define new types that include both data (fields) and behavior (methods).

```
Java
class Car {
   String model;
   int year;
}
Car myCar = new Car();
```

(b) Type Systems

A Type System is a logical framework consisting of a set of rules that assigns a property called a "type" to various constructs of a computer program (such as variables, expressions, functions, or modules).

Purposes of a Type System:

1. Error Detection

The type system identifies "type errors" where an operation is performed on a type that doesn't support it. This prevents the program from crashing or behaving unpredictably at runtime.

- **Example (Java - Static Checking):**

Java

int x = "Hello"; // Error: Incompatible types (String cannot be converted to int)

The compiler catches this before the program even runs.

2. Program Correctness

Type systems act as a form of documentation and a formal constraint. By defining types, the programmer ensures that the logic follows the intended design. It prevents logical mishaps like adding a "Date" to a "Currency."

- **Example (Python - Runtime Check):**

Python

```
def calculate_area(radius):
   return 3.14 * (radius ** 2)
calculate_area("five") # TypeError: unsupported operand type(s) for **: 'str' and 'int'
```

3. Compiler Optimizations

When the compiler knows the exact type of a variable, it can generate highly efficient machine code. For instance, knowing a variable is a 32-bit integer allows the compiler to use specific CPU instructions for integer arithmetic rather than slower, more general instructions.

- **Example (C):**

C

```
int a = 10;
int b = 20;
int c = a + b;
```

The compiler maps a and b directly to fixed memory addresses or registers, making the addition nearly instantaneous.

Task 2: Type Checking Mechanisms

Type checking is the process of verifying and enforcing the constraints of types. It ensures that operations are performed on compatible data types (e.g:preventing you from dividing a string by an integer). This process can happen either at compile-time or runtime.

(a) Static Type Checking

Definition:

Static type checking occurs during the compilation phase, before the program is executed. The compiler verifies all type safety rules by analyzing the source code. In these languages, variables usually must be declared with a specific type, and that type cannot change throughout the program's lifecycle.

Advantages:

- **Early Error Detection:** Most bugs related to type mismatches are caught before the code ever runs.
- **Performance:** Since types are determined beforehand, the compiler can generate optimized machine code without needing to check types during execution.
- **Code Completion:** IDEs can provide better "IntelliSense" or autocomplete features because they know exactly what methods are available on a specific object.

Disadvantages:

- **Verbosity:** Requires more "boilerplate" code (explicitly declaring types).
- **Rigidity:** It can be less flexible when dealing with complex data structures that might change shape.

Languages: C, C++, Java, Rust, Swift.

Code Example (Java):

Java

```
public class StaticExample {
   public static void main(String[] args) {
       int number = 10;
       // number = "Hello"; // COMPILATION ERROR: incompatible types

       System.out.println(number + 5);
   }
}
```

(b) Dynamic Type Checking

Definition:

Dynamic type checking happens at runtime (while the program is running). The interpreter associates types with the values rather than the variables. A variable can hold an integer at one moment and a string the next.

Advantages:

- **Flexibility:** Allows for rapid prototyping and shorter code since you don't need to specify types.

- **Generic Programming:** It is easier to write functions that can handle multiple types of data without complex syntax.

Disadvantages:

- **Runtime Errors:** Type errors may only surface when a specific line of code is executed, which can lead to crashes in production.
- **Execution Speed:** The interpreter must check the type of a value every time an operation is performed, which adds overhead.

Languages: Python, JavaScript, Ruby, PHP.

Code Example (Python):

Python

```python
# No type declaration needed
data = 10
print(data + 5) # Works fine

data = "Hello"  # Variable 'data' now holds a string (perfectly legal)
print(data + 5) # RUNTIME ERROR: TypeError: can only concatenate str to str
```

(c) Code Illustration & Comparison

To better understand the difference, let's look at how the two systems handle a similar scenario involving a function that expects a specific kind of input.

Static Typing Illustration (C++):

In C++, the compiler checks if the arguments passed to a function match the defined parameters. If they don't, the code **will not compile**.

C++

```cpp
#include <iostream>
#include <string>

void greet(std::string name) {
   std::cout << "Hello, " << name << std::endl;
}

int main() {
   greet("Alice"); // Works
   // greet(123);  // Error: No matching function for call to 'greet(int)'
   return 0;
}
```

Dynamic Typing Illustration (JavaScript):

In JavaScript, the function will accept anything. The error only occurs inside the function logic during execution if the operation is invalid for that type.

```
JavaScript
function greet(name) {
  // JavaScript doesn't care what 'name' is until we try to use it
  console.log("Hello, " + name.toUpperCase());
}

greet("Alice"); // Works: "HELLO, ALICE"
greet(123);     // RUNTIME ERROR: name.toUpperCase is not a function
```

Task 3: Strong vs. Weak Typing

(a) Definitions

- Strong Typing: In a strongly typed language, once a value is assigned a type, it is strictly enforced. The language does not allow operations between mismatched types unless the programmer explicitly converts them (casting). It prioritizes predictability and safety.
- Weak Typing: A weakly typed language is more "relaxed." It will often try to "guess" what the programmer wants to do by automatically converting one data type to another to make an operation work. This is known as implicit coercion.

(b) Comparative Analysis

While "Strong" and "Weak" are often seen as a spectrum rather than a binary, here is how the four requested languages generally fall:

Feature	C	Java	Python	JavaScript
Typing Strength	**Weak** (or Mixed)	**Strong**	**Strong**	**Weak**
Type Conversion	Manual (casting) is required, but pointers allow bypass.	Strict; requires explicit casting for incompatible types.	Strict; types do not change without manual conversion.	Automatic; the engine converts types on the fly.

Implicit Coercion	Low (except for numeric types).	Very Low (mostly numeric promotion).	None (adding string to int results in an error).	High (e.g., 5 + '5' becomes '55').
Safety Level	Moderate (Memory safety risks).	High	High	Low (Logic errors due to coercion).

(c) Code Examples

The primary difference between these two systems is seen in Implicit Type Coercion (Weak) vs. Type Safety (Strong).

1. Weak Typing: Implicit Type Coercion (JavaScript)

JavaScript will try to avoid throwing an error by converting types behind the scenes.

JavaScript

```
// JavaScript Example
let a = 10;      // Number
let b = "5";     // String

let result = a + b;
// Explanation: JavaScript 'coerces' the number 10 into a string "10"
// and performs string concatenation.
console.log(result); // Output: "105" (String)

let result2 = a - b;
// Explanation: Since the '-' operator doesn't work on strings,
// JS coerces the string "5" into a number 5.
console.log(result2); // Output: 5 (Number)
```

2. Strong Typing: Type Safety (Python)

Python is strongly typed even though it is dynamic. It refuses to perform operations between mismatched types, forcing the programmer to be explicit.

Python

```
# Python Example
a = 10          # Integer
b = "5"         # String
```

```
try:
    result = a + b
except TypeError as e:
    print(f"Error: {e}")
    # Output: unsupported operand type(s) for +: 'int' and 'str'

# Correct way (Explicit Conversion):
result = a + int(b)
print(result)    # Output: 15 (Integer)
```

Task 4: Case Study and Discussion (450 minutes)

Case Study: Designing a Banking System

The Role of Static and Strong Type Systems

In the world of FinTech and core banking, software is not just an interface; it is the digital ledger of record for value. A single logic error or data type overflow can result in millions of dollars of lost assets or systemic instability.[1] Consequently, when designing a banking system, the choice of a **Static, Strongly Typed** language (such as Java, C#, or Ada) is the industry standard.

1. The Criticality of Safety

Safety in a banking system refers to the prevention of "undefined behavior" and "logical corruption."[2]

Type Safety and Monetary Integrity

In a **Strongly Typed** system, the compiler enforces strict rules about what operations can be performed on specific types.In banking, we often deal with multiple "numeric" concepts that are not interchangeable. For example, a UserID is an integer, and an AccountBalance is a decimal. In a weakly typed language, a developer might accidentally add a UserID to an AccountBalance. A strongly typed system prevents this at the source-code level.

Example: Consider a function transfer(AccountID target, Money amount). If a developer tries to pass transfer(500, "100"), a strongly typed language will flag this as an error. A weakly typed language might attempt to coerce the string "100" into a number or, worse, concatenate the values, leading to a balance of "500100" instead of an addition.

Preventing Overflow and Underflow

Banking systems must handle varying scales of data—from small retail transactions to multi-billion dollar inter-bank settlements. Static typing allows developers to specify exact bit-widths for data (e.g., Int64 vs Decimal). This precision is vital for avoiding the "Integer Overflow" bugs that have historically plagued financial systems.

2. Performance: High Throughput and Low Latency

While safety is paramount, banking systems (especially high-frequency trading or real-time payment gateways) require immense performance.

The Compilation Advantage

Static typing allows the compiler to know the exact memory layout of every object before the program ever runs. In a dynamic language like Python, the interpreter must check the type of every variable *every time* it is accessed during execution.[5] In a static language, these checks are done once (at compile time).

This results in:

1. **Direct Memory Addressing:** The CPU can jump directly to the data it needs.
2. **Inlining:** The compiler can replace function calls with the actual code logic, reducing the "overhead" of jumping between different parts of the program.
3. **Predictable Garbage Collection:** In languages like Java or C#, the static nature of types makes memory management more predictable, which is essential for ensuring that the system doesn't "freeze" during a peak transaction window (e.g., Black Friday).

3. Maintainability: The Documentation of Types

Banking systems are rarely "finished." They evolve over decades, passing through the hands of hundreds of developers.[6]

Self-Documenting Code

In a large-scale banking codebase (often millions of lines of code), Static Typing acts as a form of verified documentation. When a developer encounters a function:

public TransactionReceipt processPayment(Account sender, Account receiver, CurrencyAmount amount)

The types (Account, CurrencyAmount) tell the developer exactly what the function expects and what it returns. In a dynamically typed language, the developer would have to search through documentation or other files to guess what the "sender" object actually contains.

Refactoring Confidence

When a bank needs to update its internal logic—for example, moving from a 10-digit account number to a 12-digit IBAN format—a static type system is a lifesaver. If the developer changes the type definition of an AccountNumber, the compiler will immediately highlight every single line of code in the entire system that is now "broken" by this change. This allows for massive updates with a guarantee that no hidden "type mismatches" remain in the production code.

4. Real-World Implications: The "Ariane 5" Lesson

While not a bank, the Ariane 5 rocket failure is the gold-standard case study for type safety. A 64-bit floating-point number was converted into a 16-bit signed integer. This caused an overflow, leading to the destruction of the rocket. In banking, similar "narrowing" conversions (e.g., converting a high-precision interest rate into a low-precision decimal) can lead to "salami slicing" fraud or massive rounding errors over millions of transactions. A **Strong Type System** forces the developer to handle these conversions explicitly, ensuring that precision is never lost by accident.

5. Conclusion: Why Java/C# Over JavaScript/Python for Banking?

While Python is excellent for data analysis and JavaScript is great for UI, they are unsuitable for the *core logic* of a banking system for the following reasons:

- **Ambiguity:** Banking cannot afford the ambiguity of 1 + "1" = "11".
- **Late Detection:** A bug in a Python-based banking system might not be discovered until a specific customer with a specific balance triggers a rare edge case at 3:00 AM. In Java, that bug would likely be caught before the code was even allowed to be saved.
- **Scale:** Static types allow IDEs to map out complex relationships in the software, making it possible for a team of 500 developers to work on the same banking engine without stepping on each other's toes.

For a banking system, **Static, Strong Typing** is not just a preference; it is a fundamental requirement for the security and stability of the global economy.

For the design of a **Scientific Simulation**, the architectural priorities shift significantly from business logic to computational efficiency and numerical precision. When simulating complex physical phenomena—such as fluid dynamics, quantum mechanics, or climate modeling—the choice of type system is a foundational decision that dictates the project's success.

Case Study: Designing a Scientific Simulation

The Architecture of Precision and Speed

Scientific simulations—ranging from global climate modeling and astrophysicists' N-body simulations to computational fluid dynamics (CFD) and molecular modeling—occupy a unique space in software engineering. Unlike consumer applications, where the bottleneck is often network latency or user interaction, the bottleneck in scientific computing is the hardware itself: the CPU, the memory bandwidth, and the floating-point units.

To address these constraints, a **Static and Strong** type system (exemplified by C++, Fortran, and Rust) is not just a preference; it is a fundamental requirement.

1. The Performance Imperative: Bridging the Gap to Hardware

The primary driver for using static typing in scientific simulations is the need to eliminate abstraction overhead. In a simulation where an inner loop might execute 10^{12} iterations, a single extra clock cycle per iteration results in hours or days of wasted supercomputer time.

A. Zero-Overhead Abstractions and Machine Code Generation

In a statically typed language, the type of every variable is fixed before execution. This allows the compiler to perform **Static Analysis** and generate optimized machine code.

Direct Mapping: The compiler maps variables directly to physical CPU registers. In a dynamic language like Python, the interpreter must check a "type tag" at runtime to determine if a variable is an integer or a float, then look up the appropriate addition function in a table (dynamic dispatch).[1] This "type tax" is avoided entirely in static systems.

Inlining: The compiler can perform "function inlining," where the code of a small function is inserted directly into the caller, eliminating the overhead of a function call.[2] This is crucial for mathematical kernels.

B. Leveraging Modern CPU Features: SIMD and Vectorization

Scientific computing relies heavily on **Vectorization**.[3] Modern processors feature Advanced Vector Extensions (AVX) that allow a single instruction to operate on multiple pieces of data (SIMD—Single Instruction, Multiple Data).[4]

Static Guarantees: For a compiler to safely use SIMD, it must guarantee that the memory is aligned and that the types in an array are uniform. Static type systems provide these guarantees.

Auto-Vectorization: Compilers like gcc or clang can automatically transform a standard for loop into a vectorized one if they can prove type safety at compile-time. Dynamic languages usually require manual calls to C-extensions (like NumPy) because the interpreter cannot prove that the types won't change mid-loop.

C. Cache Locality and Memory Topology

Performance in scientific simulations is often limited by the speed at which data can move from RAM to the CPU (the "Memory Wall").[5]

Data Layout: Static typing allows for "flat" memory layouts, such as **Arrays of Structures (AoS)** or **Structures of Arrays (SoA)**. These layouts ensure that the CPU cache stays "warm" with relevant data.

Avoiding Pointer Indirection: Dynamic languages often use "boxed" types, where a variable is actually a pointer to an object elsewhere in memory. This causes "cache misses" as the CPU has to hunt for data. Static systems keep data contiguous, which can increase performance by 10x to 100x in numerical tasks.

2. Mathematical Safety and Numerical Integrity

In a scientific context, "safety" is synonymous with **numerical correctness**. A simulation that runs fast but produces an incorrect result due to a unit mismatch or precision loss is worse than no simulation at all.

A. Dimensional Analysis and Strong Typing

One of the most insidious errors in scientific programming is the **unit error**. In 1999, the Mars Climate Orbiter was lost because one piece of software used English units (pound-seconds) while another used metric (newton-seconds).

User-Defined Types: Strong type systems allow us to define specific types for physical units. Instead of using double for everything, we can define Type Meter and Type Second.

Compiler Enforcement: If a developer tries to add a Meter to a Second, a strongly typed compiler will throw an error. In a weakly typed system, the language would simply add the raw numbers, leading to a physically impossible result that might go unnoticed for months.

B. Floating-Point Precision Control

Floating-point numbers are approximations.[6] In simulations that run for millions of time steps, small rounding errors accumulate (the "Butterfly Effect").

Explicit Precision: Statically typed languages force developers to be explicit: float (32-bit), double (64-bit), or long double (80/128-bit).[7]

Prevention of Implicit Downcasting: A strong type system prevents the language from silently "downcasting" a 64-bit result to 32-bit to save space, which could introduce subtle inaccuracies that corrupt the entire simulation's validity.

C. Concurrency and Data Races

Modern simulations run on thousands of CPU cores in parallel. Managing memory in these environments is notoriously difficult.

Rust's Ownership Model: Languages like Rust use the type system to enforce "Ownership" and "Borrowing" rules. The compiler ensures that two different threads cannot modify the same piece of data at the same time. This eliminates **Data Races** at compile-time without the performance penalty of a "Garbage Collector" or "Mutex locks," which are common in dynamic or less-strict static languages.

3. Maintainability: Managing Decades of Complexity

Scientific software has an unusually long lifespan. A weather model or a structural analysis tool might be maintained for 30 to 40 years.

A. The Type System as Formal Documentation

In a collaborative project involving physicists, mathematicians, and engineers, the type system acts as the **source of truth**.

Contracts: A function signature like solve_poisson(Grid3D g, BoundaryConditions b) is an explicit contract. A developer knows exactly what inputs are required. In a dynamic language, one would have to read through the entire function body to guess what properties the g and b objects must have.

B. Refactoring Large-Scale Codebases

Scientific models are frequently updated to incorporate new research.[8]

Impact Analysis: If a scientist changes a fundamental data structure (e.g., moving from a Cartesian grid to a Spherical grid), a static type system will immediately flag every single line of code in the millions-of-lines project that is affected by this change.

Verification: This allows for "fearless refactoring," where engineers can improve the code's efficiency knowing that the compiler is verifying the integrity of the data "plumbing" at every step.

4. The "Two-Language" Problem and the Hybrid Solution

While the simulation engine must be static and strong, the broader scientific ecosystem recognizes the value of dynamic languages. This has led to the **Hybrid Architecture**:

The Computational Kernels (C++/Fortran/Rust): These are the "engines" of the simulation. They are statically typed, strongly typed, and manually optimized for hardware. They handle the "heavy lifting" of matrix inversions, particle updates, and differential equation solvers.

The Steering Layer (Python/Julia): This layer is used for setting up the simulation parameters, handling file I/O, and visualizing the results. Python's dynamic nature makes it perfect for "glue code," while the static core ensures the simulation remains performant.

This hybrid approach allows scientists to have the best of both worlds: the **safety and speed** of static typing for the core math, and the **flexibility** of dynamic typing for experimental analysis.

5. Final Conclusion

In the domain of scientific simulation, the choice of a **Static, Strongly Typed** system is a matter of professional and scientific rigour. The system serves as:

A Performance Optimizer: Enabling the code to run at the "speed of the metal."

A Mathematical Guardrail: Preventing unit mismatches and precision errors that invalidate the science.

An Engineering Foundation: Allowing complex, multi-decade projects to remain maintainable and correct as they evolve.

Choosing a weak or dynamic system for the core of a simulation would result in a tool that is not only prohibitively slow on modern supercomputers but also prone to "silent errors"—the most dangerous kind of failure in the pursuit of scientific truth.

Case Study: Designing a Web Application

The Strategic Choice: Gradual Typing (Dynamic + Static)

Web applications are unique in the software world because they are accessed by diverse clients (browsers, mobile apps, IoT devices) and must integrate with an ever-evolving ecosystem of third-party APIs. Unlike a banking system (where the internal ledger is the source of truth) or a scientific simulation (where physics is the source of truth), the "truth" in a web app is often a moving target.

For this scenario, the preferred choice is a **Dynamic language** (like JavaScript) enhanced by a **Static type checker** (like TypeScript).

1. Safety: Navigating the "Boundary" Problem

In web development, the most dangerous area is the **Boundary**: the point where your application receives data from the outside world (a user submitting a form or an API returning a JSON object).

Type Coercion and Security

Weakly typed languages like JavaScript can lead to "silent failures" through implicit coercion.

- **The Problem:** If a web app calculates a shipping discount by taking totalPrice - discountCode, and the discountCode is accidentally a string "10", JavaScript might try to subtract them, but if it were an addition (totalPrice + discountCode), it would concatenate them (e.g., 100 + "10" = "10010").
- **The Safety Solution:** By using a strong/static layer like TypeScript, these operations are flagged during development. The system forces the developer to validate that the "string" from the web form is converted to a "number" before any math occurs.

Handling "Null" and "Undefined"

The "Billion Dollar Mistake" in programming is the null pointer exception. In web apps, this usually looks like: Cannot read property 'name' of undefined.

- Static type systems in web development (like TypeScript's "Strict Null Checks") force the developer to account for the possibility that a user profile or a piece of data might not have loaded yet. This drastically reduces the "White Screen of Death" errors that plague un-typed web applications.

2. Performance: Balancing Latency and Development Velocity

When we discuss performance in web applications, we must distinguish between **Runtime Performance** and **Development Performance**.

Runtime Performance (JIT Engines)

Modern web browsers use **Just-In-Time (JIT)** compilation. Engines like V8 (Chrome/Node.js) observe the code as it runs. If a function is consistently called with the same types, the engine optimizes it into machine code that rivals the speed of C++.

- **Type Stability:** Ironically, writing code that *looks* like it has static types (always passing the same types to the same functions) helps the JIT engine perform better. A static type system encourages this "type-stable" coding style, leading to faster execution.

Development Performance (Time-to-Market)

In the web industry, being first to market is often more important than having the fastest possible code.

- **Reduced Boilerplate:** Unlike Java or C++, where you must define a class for every small piece of data, dynamic languages allow for "Object Literals." This allows a web developer to create a complex response for a UI in seconds.
- **Hot Reloading:** Because dynamic languages don't require a full re-compilation of the entire system, web developers can see their changes in the browser almost instantly after saving a file. This creates a "tight feedback loop" that is impossible in heavy static systems.

3. Maintainability: Scaling with "Gradual" Types

The greatest challenge of a web app is not the first version, but the version three years later when the original team has left.

The Evolution from JavaScript to TypeScript

Small startups often start with pure JavaScript because it is fast and flexible. However, as the codebase reaches 50,000+ lines, it becomes a "spaghetti" of unknown data structures.

- **Gradual Adoption:** You don't have to type everything at once. You can start by typing the most critical parts (like payment logic) and leave the UI components dynamic. This "Gradual Typing" allows the system to grow in strictness as it grows in complexity.
- **Refactoring Safety:** If you decide to change a user.id from a number to a string (to accommodate UUIDs), a static type system will highlight every single component, API call, and utility function that needs to be updated. Without this, you would have to search and replace manually, inevitably missing a spot that causes a crash in production.

Documentation as Code

In web development, the "API Contract" is vital. When a backend developer changes the data structure, the frontend developer needs to know immediately. By sharing type definitions between the frontend and backend, the "Contract" is enforced by the compiler. If the backend changes a field, the frontend build will fail, preventing a broken version from ever reaching the user.

4. Integration and the Ecosystem

Web applications are built on top of thousands of open-source libraries (NPM packages).

- **Type Definitions (@types):** One of the reasons for the success of TypeScript in web development is the community-driven "DefinitelyTyped" project. It provides static type definitions for almost every JavaScript library in existence.
- **Interoperability:** Because the underlying system is still dynamic (JavaScript), the app can easily load "dirty" or "untyped" data from legacy systems without the entire program crashing, which is a common struggle in strictly static environments like C++ or Java.

5. Final Discussion Summary

For a **Web Application**, the choice is not a binary one between Static and Dynamic. Instead, the optimal design uses a **Dynamic Core** for flexibility and speed, wrapped in a **Static Layer** for safety and maintainability.

Feature	Design Choice: Gradual Typing
Why not pure Static?	Too rigid; slows down rapid UI iteration and integration with messy web APIs.
Why not pure Dynamic?	Too dangerous; leads to "undefined" errors and makes refactoring nearly impossible at scale.
The Result	A system that feels like a dynamic language during development but behaves like a static language during error checking.

In conclusion, while the banking system demands **Absolute Integrity** and the scientific simulation demands **Absolute Speed**, the web application demands **Absolute Agility**. A type system that allows for gradual strictness is the only way to achieve all three as a project evolves from a prototype to a global platform.