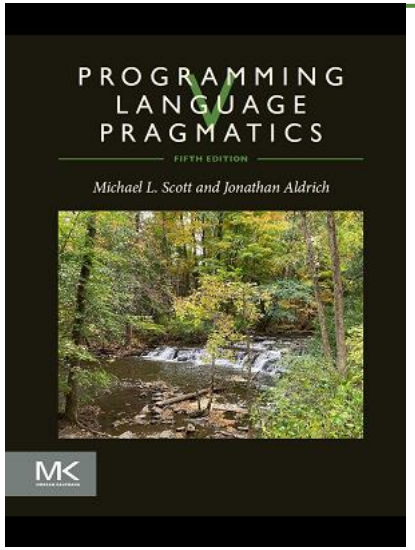
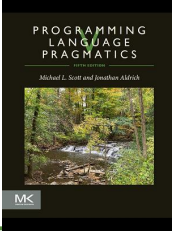
Chapter 7: Type Systems



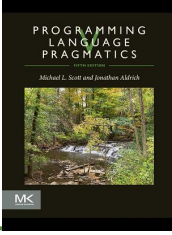
Programming Language Pragmatics, Fifth Edition

Michael L. Scott and Jonathan Aldrich



What is a type? Three views:

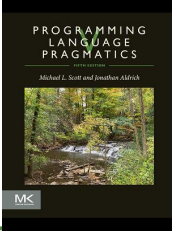
- Denotational: a collection of values from a domain
  - e.g. the 32-bit integers (int), or the real numbers representable as IEEE single-precision floats (float)
- Structural: a description of a data structure in terms of fundamental constructs
  - e.g. a Point is a record made up of fields x and y, both of type int
- Behavioral: the set of operations that can be applied to an object
  - e.g. a Stack has operations push(v) and pop()
  - Compared to structural, the focus is on operations and their behavior



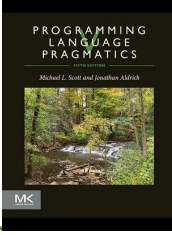
What are types good for?

- Documentation
  - What do I need to pass to this library function?
- Implicit context for compilation
  - Is this + an integer or a floating point operation?
- Checking that meaningless operations do not occur
  - e.g. “hello, world” - 5 does not make sense
  - Type checking cannot prevent all meaningless operations
    - It catches enough of them to be useful

Type terminology



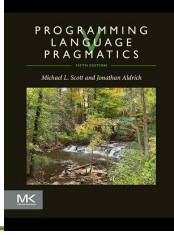
- Type safety
  - The language ensures that only type-appropriate operations are applied to an object
- Strong vs. weak typing
  - The degree to which the language enforces typing invariants and prevents accidental errors
  - More of a spectrum than a hard distinction
- Static vs. dynamic typing
  - Whether types are checked at compile time or run time



Type terminology examples

- Java is type safe, strongly and statically typed
- Common Lisp is type safe, strongly and dynamically typed
- C and C++ are statically and strongly typed, but are not (fully) type safe
- JavaScript is type safe, weakly and dynamically typed
  - JavaScript allows many implicit conversions between types, some of which are surprising
  - It is therefore weakly typed compared to the above languages

Why might JavaScript be considered weakly typed?

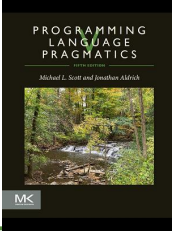


- What do you think the following expressions evaluate to?

```
[] == ![];

"b" + "a" + +"a" + "a";

null == 0;
null > 0;
null >= 0;
```

- Try them in a JavaScript interpreter! (e.g. node.js)

Examples are from https://github.com/denysdovhan/wtfjs which has a lot of other interesting ones!



JavaScript example explanations

```
[] == ![];        // true
```

- ! coerces [] to a Boolean. [] is *truthy* so ![] is false.
  JavaScript compares values at the same type.
  JavaScript converts false to 0, and [] to "" to 0.

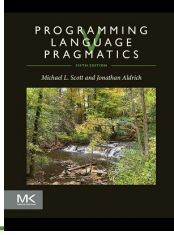
```
"b" + "a" + +"a" + "a";      // 'baNaNa'
```

- +"a" converts "a" to a number. Since a is a letter, not a sequence of digits, it is converted to NaN (not a number), which converts to "NaN"

```
null == 0;     // false
null > 0;      // false
null >= 0;     // true
```

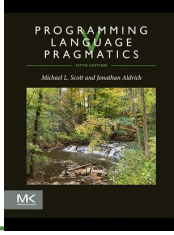
- == treats null specially. It is converted to undefined for comparison; the equality is false. The relational operators just convert both sides to numbers; null is converted to 0.

Classification of types & examples



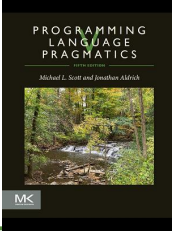
- Discrete types – (easily) countable, ordered
  - integer -1, 0 , 1, 2, …
  - boolean true, false
  - char ‘A’, ' ', '\n’, '©'
  - enumeration enum weekday { sun, mon, tue, … }
  - subrange 1..100
- Scalar types - one-dimensional
  - (all discrete types)
  - rational 22/7
  - real 3.14159

Classification of types & examples (continued)



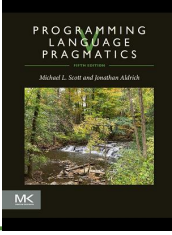
- Composite types – made up of other types
  - records { x: int, y : int }
  - datatypes/variants/unions union { street: StreetAddr, po: POBoxAddr }
  - arrays int[]
  - strings String
  - sets set(['a', 'b', 'c', 'd']), Set<String>
  - pointers int *
  - lists int list, List<Int>
  - files FILE *, File of Char

Orthogonality in type systems



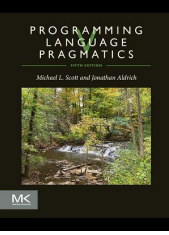
- Orthogonality is a desirable property
  - There are no restrictions on the way types can be combined
- Type theory typically studies orthogonal type constructs
  - e.g. we provide a grammar for types, they can be constructed in any way
- Most languages restrict orthogonality
  - Often for practical reasons, e.g. minimizing syntactic overhead or making type checking decidable
  - Example: ML only allows polymorphism at a let
  - Example: Java classes combine records with recursive types

Subtyping



- When one type can be safely used as another type
  - e.g. in most languages an integer can be used as a real
  - The “operational” definition of subtyping
- Other definitions
  - Intuitive: A<:B if A is a B
    - e.g. a StreetAddress is an Address
  - Denotational: A <: B if A describes a subset of the values that B describes
    - e.g. the integers are a subset of the reals
  - Structural: A <: B if A has all of the structure of B (and maybe more)
  - Behavioral: A <: B if A has all the operations that B does, and they behave as we’d expect for a B

Subtyping rules



- Subsumption - a subtype can be treated as a supertype:

$$\frac{\Gamma \vdash e : \tau_1 \quad \tau_1 \leq \tau_2}{\Gamma \vdash e : \tau_2} \; \textit{T-subsume}$$

- Subtyping is reflexive and transitive:

$$\frac{}{\tau \leq \tau} \; \textit{S-reflexive}$$

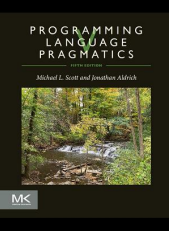
$$\frac{\tau_1 \leq \tau_2 \quad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3} \; \textit{S-transitive}$$

- We can capture some of Java's other subtyping rules as follows:

$$\frac{}{\texttt{int} \leq \texttt{long}} \; \textit{S-int-long}$$

$$\frac{}{\texttt{long} \leq \texttt{float}} \; \textit{S-long-float}$$

$$\frac{}{\texttt{float} \leq \texttt{double}} \; \textit{S-float-double}$$



In-class exercise: typing derivations with subtyping

- Construct a derivation that types the expression 1 + 2.5
- You can use the following rules in your derivation:

$$\frac{\Gamma \vdash e : \tau_1 \quad \tau_1 \leq \tau_2}{\Gamma \vdash e : \tau_2}\ \textit{T-subsume}$$

$$\frac{}{\texttt{int} \leq \texttt{long}}\ \textit{S-int-long}$$

$$\frac{}{\tau \leq \tau}\ \textit{S-reflexive}$$

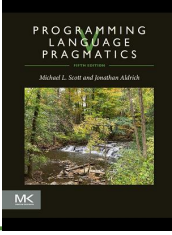
$$\frac{}{\texttt{long} \leq \texttt{float}}\ \textit{S-long-float}$$

$$\frac{\tau_1 \leq \tau_2 \quad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3}\ \textit{S-transitive}$$

$$\frac{}{\texttt{float} \leq \texttt{double}}\ \textit{S-float-double}$$

$$\frac{\Gamma \vdash e_1 : \texttt{double} \quad \Gamma \vdash e_2 : \texttt{double}}{\Gamma \vdash e_1 + e_2 : \texttt{double}}\ \textit{T-add-double}$$

(press pause for more time)



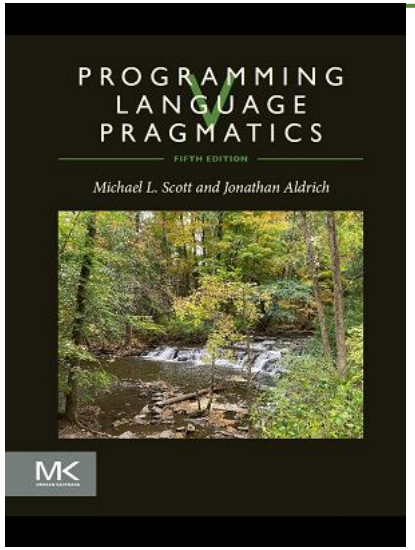
In-class exercise: solution

- Construct a derivation that types the expression 1 + 2.5

Answer: (one rule name is left out for brevity)

```
                                                  ---------------- S-long-float     -------------------- S-float-double
                                                  long <= float                     float <= double
                                  -------------   ------------------------------------------------------------ S-transitive
                                  int <= long     long <= double
------------ T-const              ----------------------------------- S-transitive
• |- 1 : int                      int <= double
---------------------------------------- T-subsume      -------------------- T-const
• |- 1 : double                                          • |- 2.5 : double
------------------------------------------------------------------------------------ T-add-double
                                  • |- 1 + 2.5 : double
```

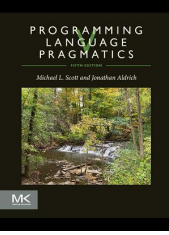
Section 7.2: Type Checking



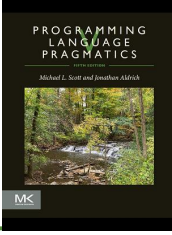
Programming Language Pragmatics, Fifth Edition

Michael L. Scott and Jonathan Aldrich

Aspects of type checking



- Type compatibility: when can a value of type A be used in a context that expects type B?
  - Similar to our first definition of subtyping
  - But type compatibility sometimes differs from the natural subtyping relation
    - e.g. C also allows long values to be implicitly truncated when assigned to an int variable
    - The designers of C felt this was convenient, but it can cause errors
    - To help programmers avoid these errors, Java type compatibility does not allow this, though programmers who want this behavior can do it with an explicit conversion.
- Type equivalence: when two types are the same
- Type inference: what is the type of an expression, given the types of the operands?



Structural vs. name equivalence: motivation

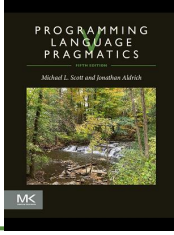
- Are these equivalent?

```
struct person {
  string name;
  string address;
}
struct school {
  string name;
  string address;
}
```

- Some languages let you choose. E.g. in Ada:

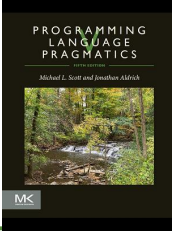
```
type Score is integer;                  // structural equivalence
                                        // can assign between Score and integer

type Fahrenheit is new integer;         // name equivalence
type Celsius is new integer;            // can't assign Fahrenheit to Celsius
```

Structural vs. name equivalence



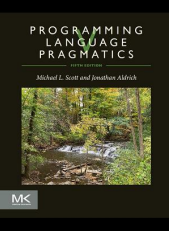
- Name equivalence is based on declarations
  - Advantage: captures the programmer's intent
  - Typical in imperative & OO languages
- Structural equivalence is based on a structural correspondence between the parts of those declarations
  - Advantage: more flexible
  - Disadvantage: can "accidentally" equate types
  - Common in functional languages
    - but they usually also have ways to support nominal equivalence

Structural equivalence



- Structural equivalence depends on simple comparison of type descriptions, substituting out all names
  - expand all the way to built-in types
- The original types are equivalent if the expanded type descriptions are the same

Coercions

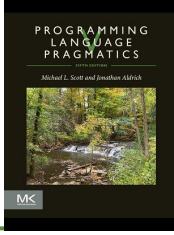


- When an expression of one type is used in a context where a different type is expected, one normally gets a type error
- But what about:

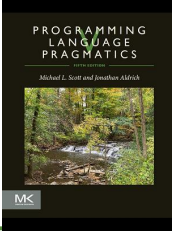
```
var a : integer; b, c : real;
        ...
c := a + b;
```

- Many languages allow things like this, and *coerce* an expression to be of the proper type
- Coercion can be based on just the types of operands, or can take into account the expected type from the surrounding context

Coercions in C



- C has lots of coercions, with fairly simple rules:
  - all **`float`**s in expressions become **`double`**s
  - **`short`**, **`int`**, and **`char`** become **`int`** in expressions
  - if necessary, precision is removed when assigning into LHS

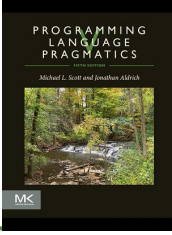


Conversions and coercions

- If you need to convert between types, but the language does not implement a coercion, you can put in a *conversion*

```
long time1 = System.currentTimeMillis();
...
long time2 = System.currentTimeMillis();
int difference = (int) (time2 – time1);          // requires a conversion (or cast) in Java
```

- Terminology
  - Type conversions (explicit, written by the programmer)
    - In C and derived languages, the word 'cast' is often used for conversions
  - Type coercions (implicit, inserted by the compiler)



Elaboration: inserting coercions

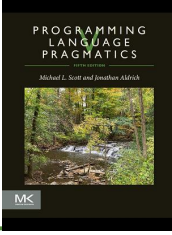
- Coercion and conversions are added in an *elaboration* pass within the compiler
  - Elaboration makes implicit things explicit

- Coercions are inserted when subsumption is used but the types have different representations
- Conversions are inserted where the user adds casts

$$\frac{\Gamma \vdash e : \textbf{int}}{\Gamma \vdash e \rightsquigarrow \texttt{float}(e) : \textbf{real}} \textit{ coerce-real}$$

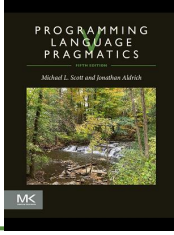
$$\frac{\Gamma \vdash e : \textbf{real}}{\Gamma \vdash (\textbf{int})e \rightsquigarrow \texttt{trunc}(e) : \textbf{int}} \textit{ convert-int}$$

Code generation for conversions and coercions



- No code is generated if types have the same representation and the provided type is a subtype of the expected type
  - e.g. converting an int to a long in Java
- A check is generated when the provided type is *not* a subtype of the expected type
  - e.g. converting an integer to a subrange 1..10
  - Some language (unsafely) skip this check, e.g. conversions from long to int in C or Java
- Conversion code is generated if the types have different representations
  - e.g. converting an int to a float

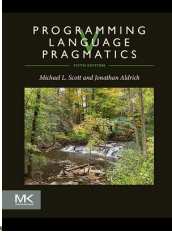
Check your understanding: structural vs. name equivalence



- Discuss the comparative advantages of *structural* and *name* equivalence for types

- (press pause for more time)

Check your understanding: structural vs. name equivalence

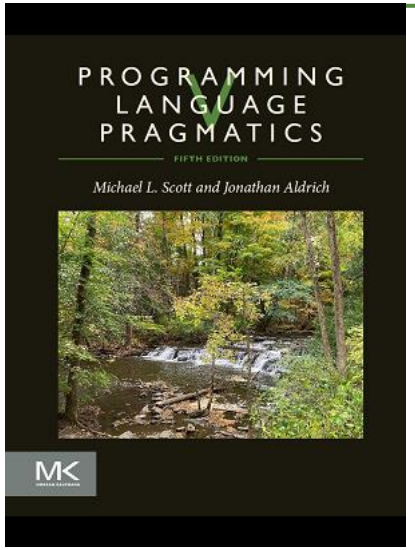


- Discuss the comparative advantages of *structural* and *name* equivalence for types

- Answer: Name equivalence can make distinctions between different types even if they have the same representation in order to capture the programmer's intent.

  Structural equivalence provides flexibility by equating types with the same structure, but when that is not intended, the type system will miss coding errors that name equivalence would catch.

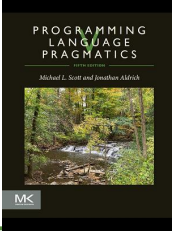
Section 7.3-7.4: Polymorphism and type inference



Programming Language Pragmatics, Fifth Edition

Michael L. Scott and Jonathan Aldrich

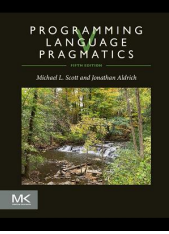
Polymorphism / generic types



- Polymorphism allows one function to work with multiple types
- Example: Polymorphism in Java

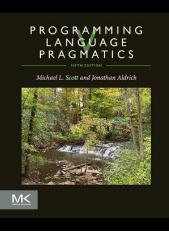
```
static <T> bool isMember(T value, T[] array) {
      for (int i = 0; i < array.length; ++i)
            if (array[i].equals(value)) return true;
      return false;
}
Integer[] a1 = { 1, 2, 3 };
String[] a2 = { “hello”, “world” };
bool result = isMember(5, a1);          // returns false
bool result2 = isMember(“hello”, a2); // returns true
bool error = isMember(5, a2);           // type error
```

Generics also support polymorphic data structures




```
template<class item, int max_items = 100>
class queue {
    item items[max_items];
    int next_free, next_full, num_items;
public:
    queue() : next_free(0), next_full(0), num_items(0) { }
    bool enqueue(const item& it) {
        if (num_items == max_items) return false;
        ++num_items; items[next_free] = it;
        next_free = (next_free + 1) % max_items;
        return true;
    }
    bool dequeue(item* it) {
        if (num_items == 0) return false;
        --num_items; *it = items[next_full];
        next_full = (next_full + 1) % max_items;
        return true;
    }
};
...
queue<process> ready_list;
queue<int, 50> int_queue; // 50 elements instead of the default 100
```

A model of generic types



- Identify function in an idealized language with generics:

```
let identity = function<T>(x:T):T {   // identity has type ∀T . T → T
    return x;
}

in print(identity<int>(5));           // prints 5
```

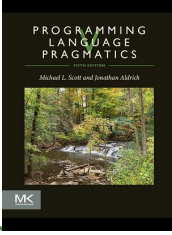
- We can extend an expression language to define and call functions of generic type:

$$e \longrightarrow \ldots \mid \texttt{function<}T\texttt{>(}x : \tau\texttt{):}\tau\ \texttt{\{ return }e\texttt{; \}} \mid e\texttt{<}\tau\texttt{>(}e\texttt{)}$$

- We extend the types to include a “forall” type, binding a generic type parameter T that can be used in the argument and result types:

$$\tau \longrightarrow \texttt{int} \mid \ldots \mid \forall T\,.\,\tau \rightarrow \tau \mid T$$

A typing rule for polymorphic function definitions



When type checking the function body expression e_2, we assume T is a type variable and the argument x_1 has type τ_1.

The function body must typecheck at type τ_2

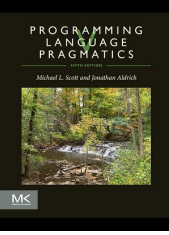
The type τ_1 must be *well-formed*, meaning it only uses type variables that are in scope

$$\frac{\Gamma,\ T{:}type,\ x_1{:}\tau_1 \vdash e_2 : \tau_2 \quad \Gamma,\ T{:}type \vdash \tau_1\ wf}{\Gamma \vdash \texttt{function<}T\texttt{>}(x_1 : \tau_1) : \tau_2\ \texttt{\{ return}\ e_2\texttt{; \}} : \forall T\,.\,\tau_1 \to \tau_2}\ t\text{-}generic\text{-}fn$$

Here is the function we are type checking

The function's type expresses that for all type parameters T, the function takes an argument of type τ_1 and returns type τ_2

A typing rule for calling polymorphic functions



This premise gets the type of the function by typechecking the expression e_2

The type argument must be well-formed

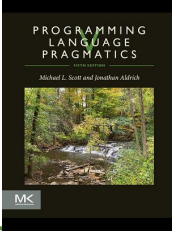
The type of the argument expression must match the function parameter's type, after substituting type argument τ for the type variable *T*.

$$\frac{\Gamma \vdash e_2 : \forall T \,.\, \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash \tau \; wf \quad \Gamma \vdash e_1 : [\tau/T]\tau_1}{\Gamma \vdash e_2\langle\tau\rangle(e_1) : [\tau/T]\tau_2} \; t\text{-}instantiate\text{-}fn$$

e_2 is a function that we are calling with type argument τ. Note that most languages can infer τ automatically, so the programmer doesn't have to write it; we show it explicitly here for clarity.

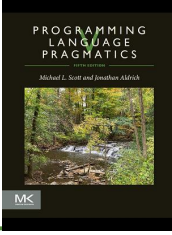
The type of the function call is the return type of the function, with the type argument τ substituted for the type variable *T*.

Implementing generics



- In C++ and Ada, the entire generic function is copied, substituting the type argument for the type variable
  - This duplicates code and can create large binaries, but can be fast as the code can be optimized with the particular type argument in mind
  - Typechecking is delayed until the copy is made—this adds flexibility but means ill-typed functions produce errors only when they are called
- Java reuses the same code for all calls to the generic function
  - Data generic type is handled indirectly through pointers
  - The function can be typechecked separately from calls to it
- C# combines these
  - Typechecking is separate; code is shared for reference types, but copies are made for every instantiation with primitive types

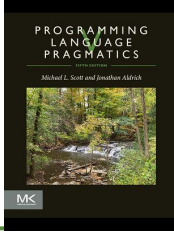
Generic parameter constraints



- Sometimes a function needs to make assumptions about the generic type
- This Java sort function assumes the type argument is Comparable:

```
public static <T extends Comparable<T>> void sort(T[] A) {
   …
   if (A[i].compareTo(A[j]) >= 0) …
   …
}
…
Integer[] myArray = new Integer[50];
…
sort(myArray);  // the typechecker verifies that type Integer implements Comparable
```

Local type inference

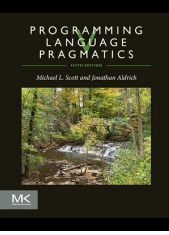


- In C++ (and many other languages):

auto x = 3.5+1;

- x will have type **double** since the right-hand side expression has that type

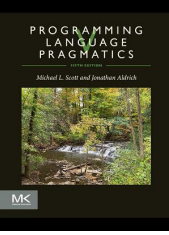
Global type inference



- In functional languages like ML or Haskell, the compiler can infer the types of functions
- Let's explore the intuition behind type inference using a simple Fibonacci function:

```
1 -- fib :: int -> int
2 let fib n =
3   let rec helper n1 n2 i =
4     if i = n then n2
5     else helper f2 (n1 + n2) (i + 1) in
6   helper 0 1 0;;
```

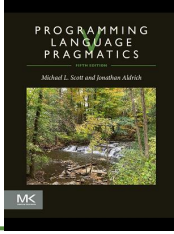
Global type inference – intuition behind the algorithm




```
1 -- fib :: int -> int
2 let fib n =
3   let rec helper n1 n2 i =
4     if i = n then n2
5     else helper n2 (n1 + n2) (i + 1) in
6   helper 0 1 0;;
```

- i : int, because it is added to 1 at line 5
- n : int, because it is compared to i at line 4
- all three args at line 6 are int consts, so n1 : int and n2 : int
  - also, the 3rd argument is consistent with the known int type of i
  - the types of the arguments to the recursive call at line 5 are similarly consistent
- since helper returns n2 (known to be int) at line 4, its return type must be int
  - and the result of the call at line 6 will be int
- since fib immediately returns this result as its own result, the return type of fib is int

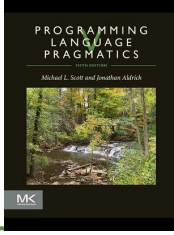
Check your understanding: generic type implementation



- Compare the implementations of generic functions in C++ and Java, and describe tradeoffs between them

- (press pause for more time)

Check your understanding: generic type implementation

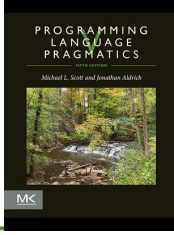


- Compare the implementations of generic functions in C++ and Java, and describe tradeoffs between them

- Answer: C++ creates a copy of a generic function for every different type the function is called with. This adds flexibility to the type system and can enable code optimizations, but it can also cause code bloat, and type errors may not be caught until a function is instantiated.

  Java avoids copying generic function code, giving up some flexibility and optimization opportunities, but keeping the code short and catching type errors in generic functions when they are written instead of at instantiation time.

Type Systems



- Types provide compiler-checked documentation, aid compilation, and catch errors
- Subtyping determines how values can flow between types
- Nominal vs. structural type equivalence is a tradeoff
- Coercions and conversions move values between types
  - checking and transforming the values as needed
- Generic types provide flexibility along with safety

- For more in-depth content
  - Like & subscribe to my channel
  - Get a copy of our book!