Basic Types and Algebra of Types

Type systems

Each programming language has its own **type system**, i.e., the information and rules that govern its types and their values—called **inhabitants** of their type. More precisely, a type system includes:

1. a set of **basic types**;
2. mechanisms to **define new types**;
3. mechanisms that **compute** on types, which include:
    1. **equivalence rules**, which specify when two types correspond to the same type;
    2. **compatibility rules**, which specify when one can use a type in place of another one;
    3. **type inference** rules/techniques, which specify how to assign a type to an expression, starting from the information on its components;
4. the specification of whether to check type constraints **statically** or **dynamically**.

Basic Types

Basic (primitive/simple/scalar) types are all types that define denotable values of the language. Common denotable values are `42`, `3.14`, and `'A'` and their (basic) types can be respectively called `int`, `float`, and `char`. We use the verb "can", because values, and thus their types, are not equally understood by all languages.

How we characterise `42` and `int` depends on the language we are considering. In Java, we would use 4 bytes to represent 42 and the type `int` would range over all whole numbers from -2^{31} to $2^{31} - 1$. In Rust, we would need to further specify what "kind" of integer we are interested in storing, e.g., the type `i32` would give us a size and range similar to that of Java, while the type `u32` would only consider positive numbers (thus the range 0 to $2^{32} - 1$).

As languages adopt different syntaxes, they also provide different ways to declare basic types

- Java adopts the syntax `type variableName`
- Rust provides the statement `let variableName : type`

Unit (vs Void) type

The most basic type is the one containing just one element (the set is a singleton).

The only inhabitant of the **Unit** type is the singleton `unit` (also represented as `()`) and it is usually associated with operations whose return type is non-usable (e.g., because they acted via some side effect, like printing on the screen). This is consistent with a mapping of the input (whatever that is) to the same, inscrutable output (the unit).

Languages like Java and C have a similar concept with the **`void`** type which, however, holds some differences with `Unit`. For example, while we can pass the `unit` as the argument of operations, we cannot obtain nor pass a `void`: as the name says, it represents emptiness and one cannot define other types using `void` subcomponents. This discrepancy with Unit becomes visible, e.g., in Java generics, which required the introduction of the `Void` type of which `null` (instead of `void`) is the only inhabitant.

Boolean type

Booleans denote the type of logical values and usually include:

- values: the two values of truth, true and false;
- operations: the main logical operations, such as conjunction (`&`), disjunction (`|`), negation (`!`), equality (`==`), exclusive or (`^`), etc.

Where present (e.g., ANSI C does not have one such a type), its values are denotable, expressible, and storable. Interestingly, while one would assume one bit would suffice to store booleans, the actual memory representation depends on the language hardware model, the architecture's basic addressable unit, and other alignment requirements.

For example, in Rust the variables of type `bool` require one byte. In (the) Java (Virtual Machine) `bools` require 2 bytes (8 as header, 1 for the value, and 7 of padding).

Character type

Characters denote all characters from a given (and defined at the language level) set of symbols and usually include:

- values: a set of character codes, e.g., two common sets are ASCII and UNICODE;
- operations: highly dependent on the language; we usually find equality (==), comparisons (<,>).

The values are denotable, expressible, and storable and, usually, in-memory representation consists of one byte (ASCII) or two bytes (UNICODE).

Integer type

Integers denote some range of whole numbers and usually include:

- values: a finite subset of integers, normally fixed at the time of definition of the language and, depending on the storage byte-size for the representation r, ranging $[2^{r-1}, 2^{r-1}-1]$ for signed (two-complement representation) and $[0, 2^{r}-1]$ for unsigned ones. Some languages have built-in support for integers of arbitrary length;

- operations: equality (==), comparisons (<,>), and the main arithmetic operations (+, -, *, /, %).

The values are denotable, expressible, and storable.

Real type

Reals denote some range of real numbers and usually include:

- values: a finite subset of reals, normally fixed at the time of definition of the language. Mainly stored either via a **fixed-point** or **floating-point** representation. Both represent reals separating their integers and decimals.
  - Fixed point numbers reserve specific bits for the integers and the decimals. Using a n-byte signed format, with f out of n bits for decimals we range $[-2^{n-1}/2^f, 2^{n-1}/2^f]$ with numbers at a constant distance of $1/2^f$.
  - Floating point numbers use the format $s \cdot m \cdot b^e$, where s is the sign (omitted when unsigned), m is the number (mantissa), b the base, and e is the exponent that places the float. The IEEE 754 format defines two formats, both with $b = 2$, but with single (8-byte) and double (11-byte) precision/exponent.
- operations: equality (==), comparisons (<,>) and the main arithmetic operations (+, -, *, /, %).

The values are denotable, expressible, and storable.

Enumeration types

An enumeration type consists of a finite set of constants, each characterised by its own name.

C, Rust, and Java (and other languages) provide all the same syntax, e.g.,

```
enum RogueOne { Jyn, Cassian, Chirrut, K2SO, Bodhi, Baze }
```

which introduces a new type named `RogueOne` consisting of a set of 6 elements, each marked by its own name. The operations available on `enums` consist of comparisons and a mechanism to obtain all values or pass from one to the next. From a pragmatic point of view, enums have two benefits: 1) they help readability, since the names of the values constitute a clear form of self-documentation of the program and 2) they let the type checking verify that an enumeration-typed variable takes only the correct values.

Not all languages integrate `enums` in a safe way, e.g., in C `enum RogueOne { Jyn, … }` is syntactic sugar for

```
typedef int RogueOne; const RogueOne Jyn=0, Cassian=1, …;
```

which equate `integers` to RogueOnes and prevents the distinction (and check the correctness) between them.

Extensional vs Intensional types

`Integers` (and `Floats` and `Chars`, ...) and `Enumerators` have one important difference: The user specifies `Enumerators` in an **extensional** way, i.e., they list all possible inhabitants of that type. On the contrary, languages specify `integers`, `floats`, etc. **intensionally**, i.e., by means of *predicates* that define their membership over some domains of possible values (e.g., 32-bit integers, floating point numbers, UNICODE chars).

The rationale is to use intensional definitions when we have a defined set of properties that identify only the inhabitants (valid values) of the type we are defining—with the pro of saving memory if the set of inhabitants is large and making the definition possible, in case of infinite sets. On the other hand, extensional definitions are useful when we do not have a clear set of rules that define the inhabitants of the type (e.g., an intentional way to define our RogueOne type could be through a rule like “the main 6 characters of the movie RogueOne”, or not 😐?)

Composite types

Enumeration types (à la C) surreptitiously introduced a new concept: we can create new types by **composing** the basic ones.

In C enumerations, we made **named sets of elements**, which correspond to integers, but other structures are possible, among which the most basic are: **arrays**, **sets**, and **pointers**.

Array types

An **array** type denotes a collection of elements of some type, each indexed by at least one **identifying key** of some type (when 2 or more keys are involved, we talk about multidimensional arrays, e.g., matrices, datacubes, etc.).

The most common notion of arrays assumes keys as non-negative integers within an interval (usually considering the $[0, n]$ range for $n + 1$ elements, which simplifies its layout in memory) and let the user define the type of the elements. Other forms of arrays, usually called maps or **associative arrays**, let the user fix both the types of the keys and the elements.

Array types

Let us see the syntax of C, Java, and Rust for declaring a linear array of integers:

`int x[3]`	`int[] x`	`let x: [i32;3]`
`x[0] = 0`		
`int x[3] = {0,0,0}`	`x = new int[3]`	`x = [0,0,0]`

Notice that that both C and Rust fix in the type declaration the size of the array (3), while Java abstracts from it in types and leaves the initialisation define the size of the array (more on this later).

Most languages (also C, Java, and Rust) extend linear-array declarations to multi-key ones

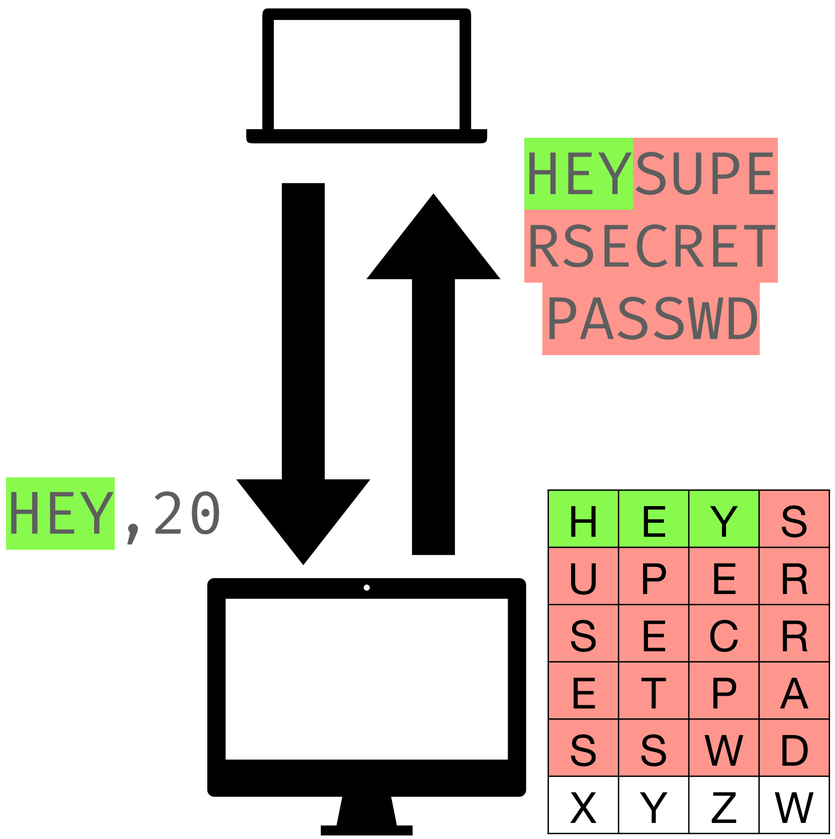
`int x[10][10]`	`int[][] x`	`let x: [[i32;10];10]`

While C, Java, and Rust coalesce the concept of multidimensional arrays and array of array (on the latter), some languages (e.g., Pascal) keep these separated.

Array types

The simplest operation on an array is the **selection** of an element by means of its index value. The most common notation (C, Java, Rust) is `a[e]` where a is the variable of type array and e is an expression. For multidimensional arrays, common syntaxes are `a[ e ][ e ][ e ]` or `a[ e, e, e ]` — the second is for languages that have both multidimensional and array of arrays. Other whole-array operations are e.g., **assignment** (=), **comparisons** (==,<,>), and arithmetic operations (performed **pairwise**).

Since they know the index type of arrays, safe languages verify that **every access to an element of the array really takes place within its "limits"** (as it does not make sense to access non-existing elements). Except some special cases, this check can only occur at runtime, which is where safe languages put appropriate checks at each access. Languages like Java and Rust guarantee this invariant at runtime (raising an error/exception when violated) but C does not. While these checks slow (a bit) programs, they prevent **buffer-overflow** attacks.

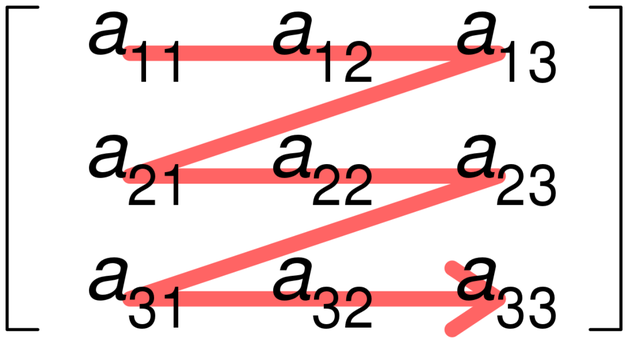


Array types

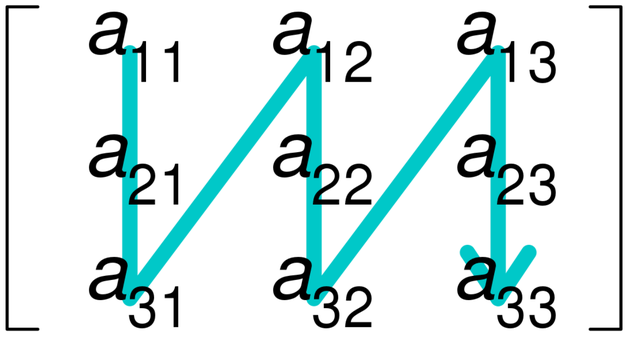
An array is usually stored in a contiguous portion of memory. For a one-dimensional array, the allocation follows the order of the indices. For multidimensional arrays, there are mainly two techniques, called **row-major** and **column-major order**.

In row order, two elements are contiguous if they differ by one in the last index. In column order two elements are contiguous if they differ by one in the first index. The row order is a little more common than the column one, mainly because row-wise accesses are more common than column-wise ones. Indeed, the locality principle of cache-miss loading favours row-wise sweeping algorithms on row-major orders and, vice versa, column-wise ones on column-major order.

Row-major order



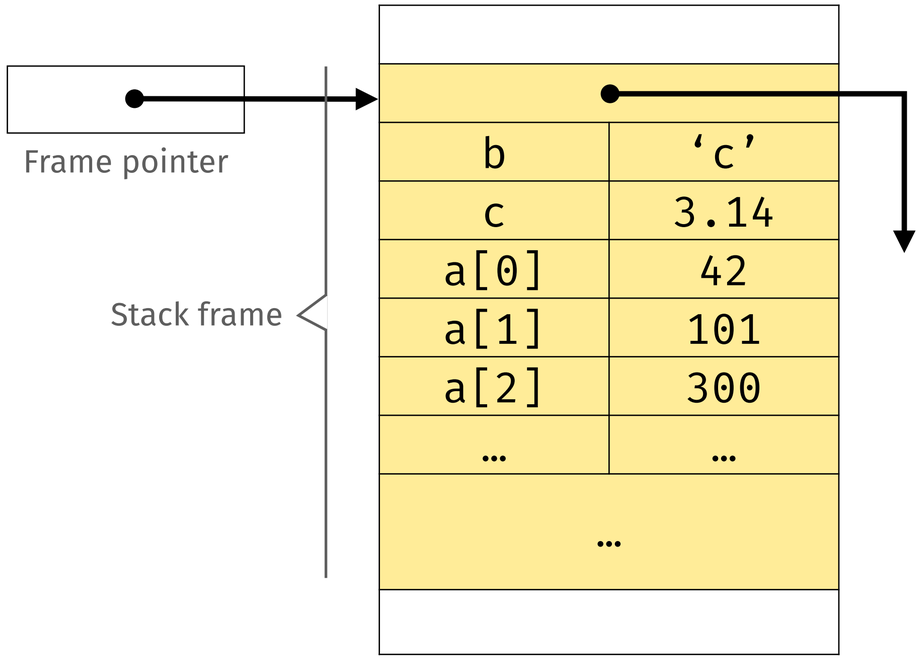
Column-major order



Array types

The **number of dimensions** and **their intervals** determine the **shape of an array**. An important aspect of a language definition is deciding **whether and when to fix the shape** of arrays. If the shape is fixed, we can either decide to define it at **compilation** (for compiled languages) or **when we process it declaration** (at runtime). Alternatively, we can have **dynamic arrays** whose shape is determined and change at runtime.

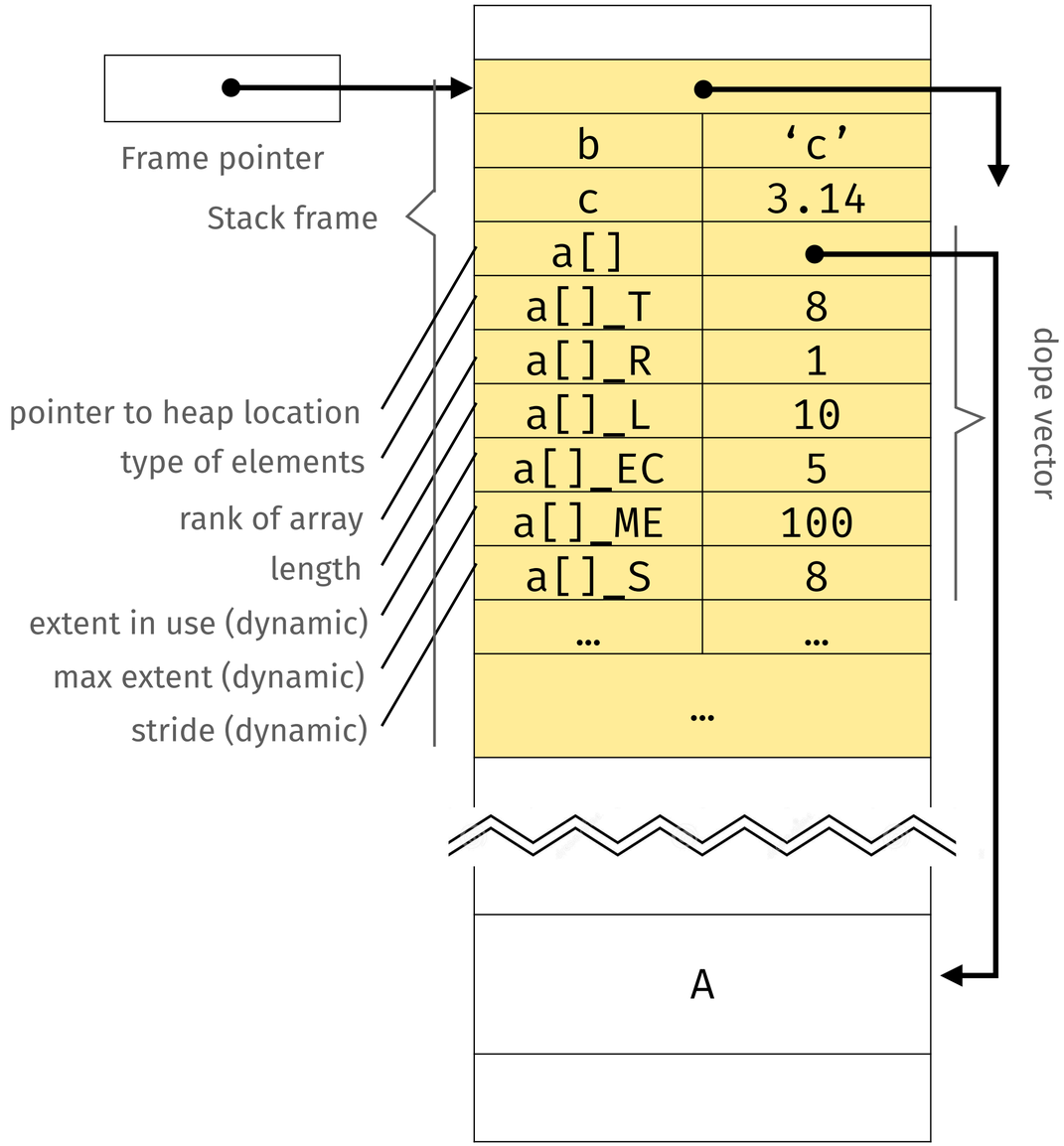
If we decide to define the array at compilation time, also called "in **static form**", we can store it in the stack frame of the block that carries its definition. In this case, we know the size needed to store the array (the offset between the first and the last items of the array), so, accessing an element of the array is similar to accessing variables of scalar types (save for some calculations needed, e.g., when accessing multidimensional arrays).



Array types

If we decide to define the array **when we process its declaration** we will know its (fixed) shape at the moment when the control reaches the declaration of the array. An example of this is, e.g., if the size depends on the value of some variable.

We can allocate the array in the stack frame of the block that carries its definition. However, since we know the size of the array only when we load the frame, we cannot safely preallocate space in the stack —a wrong estimation would either waste memory or overlap with other static variables. To work around this problem, we use the **heap** and store in the frame the pointer to the beginning of that memory region. The descriptor of such an array goes by the name of **dope vector**, also used in the case of **dynamic** arrays (with some additional items in the dope vector to track its state).



Difference among C, Java, and Rust Array types

Let us see the syntax of C, Java, and Rust for declaring a linear array of integers:

`int x[3]`	`int[] x`	`let x: [i32;3]`
`x[0] = 0`		
`int x[3] = {0,0,0}`	`x = new int[3]`	`x = [0,0,0]`

In C, the declaration corresponds to the (static) allocation of the array, which we can use right away.

In Java, we do not create the array when we declare its variable, but (like any non-primitive Java type) the name is a **reference** to some “array of integer” value. E.g., at line 3, we assign to x a **new** array (in the heap).

Also in Rust, the declaration only introduces the annotation of the name x, which we later bind to a (static) array (line 3). In this case, the type carries to the assignment to check the size constraint.

Set types

A set type denotes a flat, **orderless** data structure with **unique values** of the **same type**.

The possible operations on sets include **testing inclusion** and common set manipulation operations: union, intersection, difference, and complement.

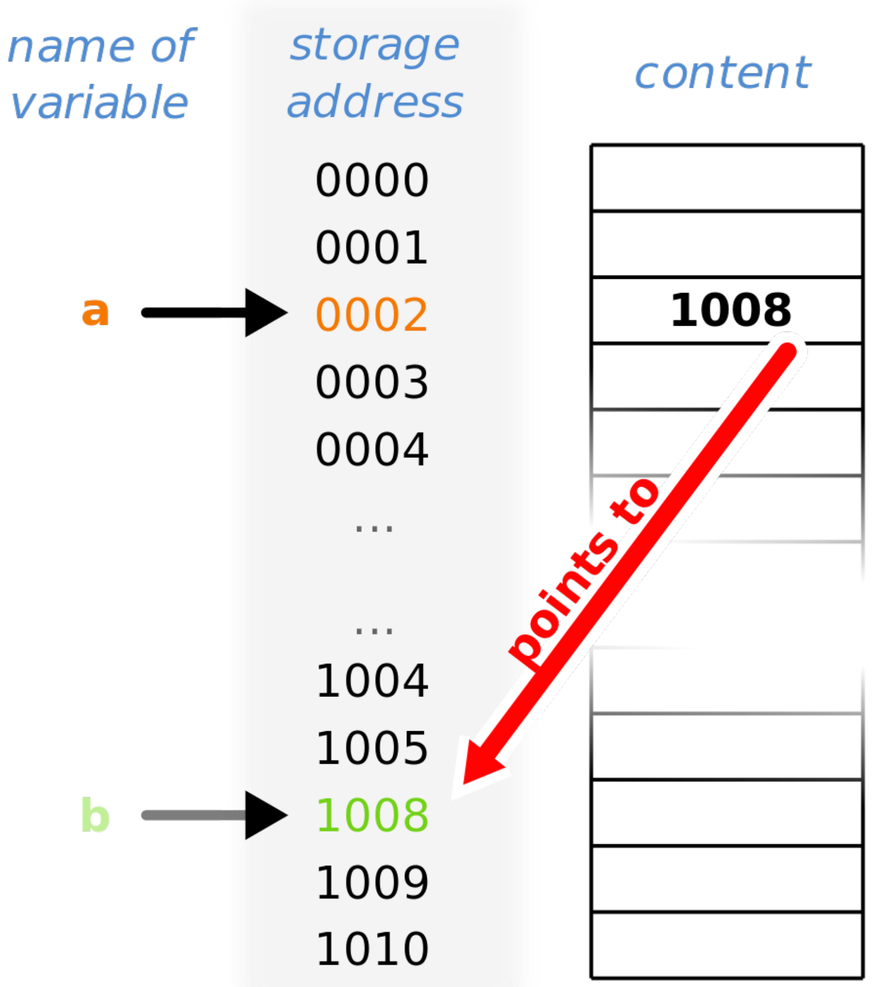
An efficient way to represent a set is by means of a bit array of length equal to the cardinality of the base type. This array is called **characteristic** and, in it, the j^{th} bit indicates whether the j^{th} element of the base type (given a standard ordering) belongs in the set. This representation allows efficient execution of set operations (bitwise operations on the physical machine), however it is unsuitable for large subsets of basic types. To work around this problem, languages often either limit the types one can use as the base types of a set or they choose alternative representations, e.g., via hash tables, trading some speed off the sake of memory.

Reference types

A reference gives indirect access to some other value (e.g., possibly assigned to some variable), i.e., they ***refer*** some datum. The typical operations supported by references are **creation**, **equality check**, and **dereferencing**, i.e., accessing the referenced datum. References are particularly present in low-level languages, where they are used to pass/share large or mutable data.

The most common implementation of references is that of the **physical address**, the **pointer**, of the datum in memory. However, pointers are just one instance of references, which can be e.g., indexes into arrays. **References can refer references**, as in data structures like trees and lists.

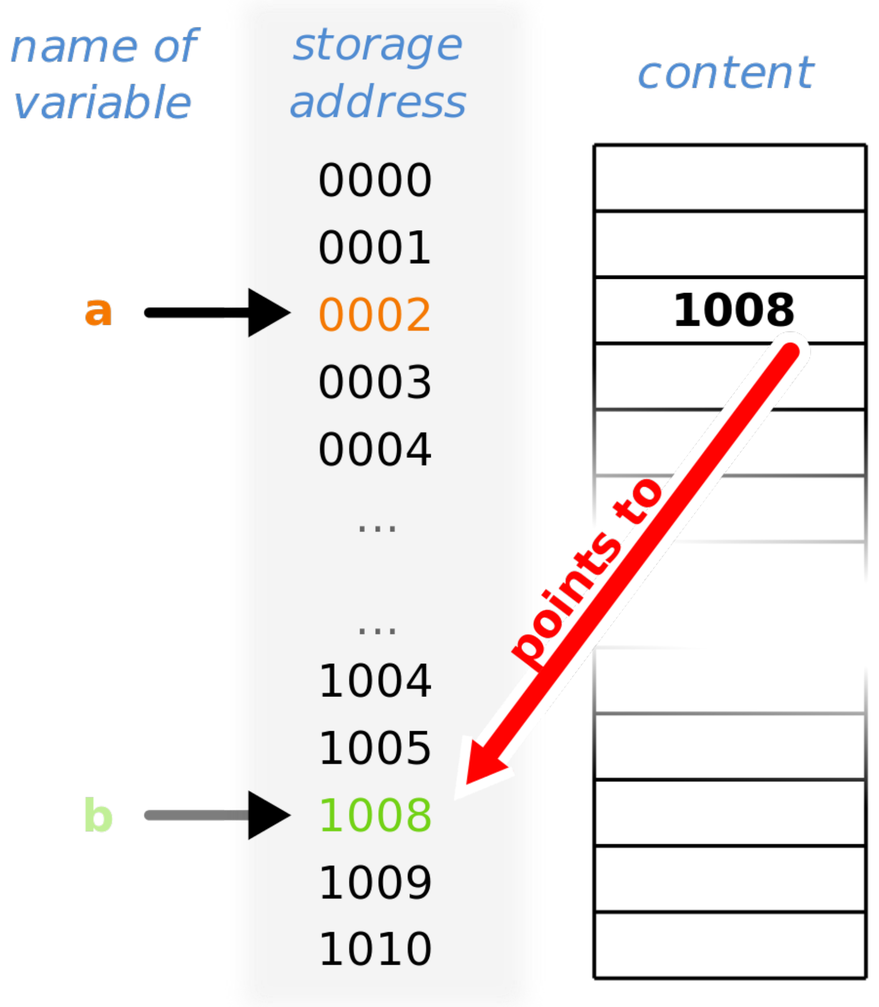
References can introduce complexity in programs, since they **ask** the programmer to **think in dynamic** rather than static **terms**. Without due diligence or dedicated checks, references can become "**wild**" (uninitialised references whose access can cause unexpected behaviour) or "**dangling**" (when their datum has been deallocated and access can lead to unexpected behaviour, especially when compatible data overwrites the deallocated datum).



Reference types

Languages with a **reference variable model** rarely provide reference types, since every variable is always a reference (e.g., Java).

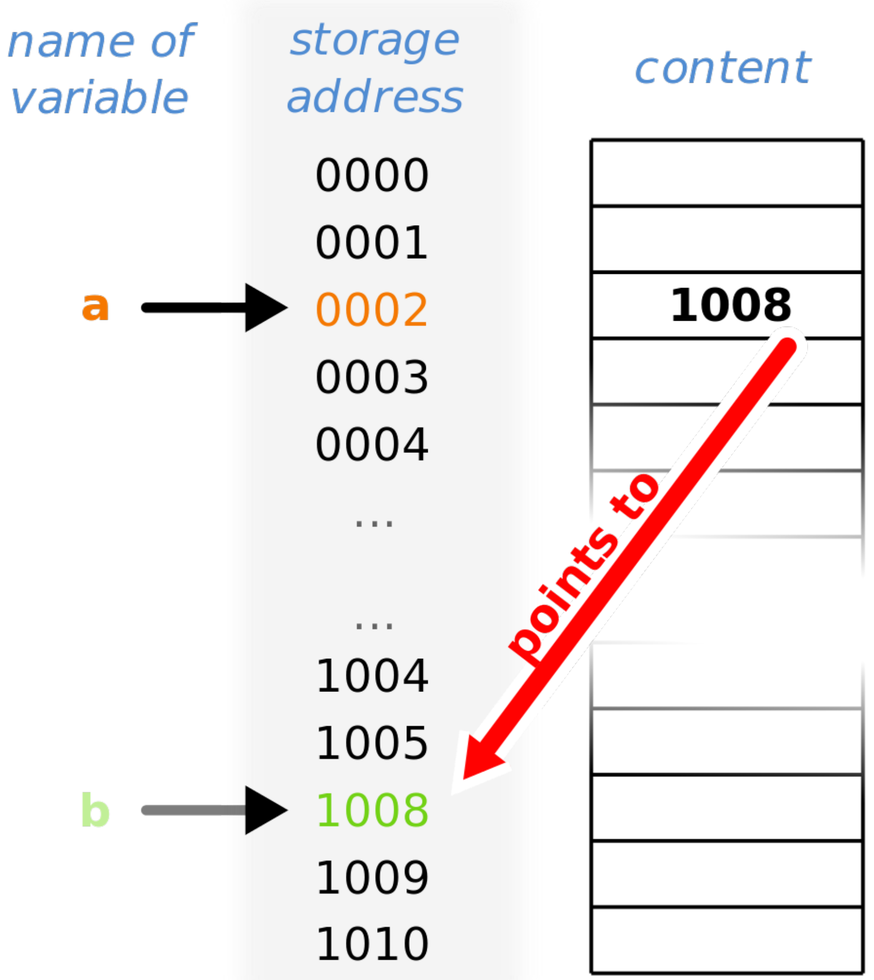
Languages with modifiable variables provide references that let the programmer refer values without dereferencing them automatically.



Reference types

In C, `int* x` specifies a reference (pointer) to a memory location (i.e., editable variables) that contain a value of type integer.

Depending on the language (model), pointers can refer arbitrary locations or follow some constraints. E.g., Pascal requires pointers to refer values allocated on the heap, while C admits pointers that refer the stack or the global area.



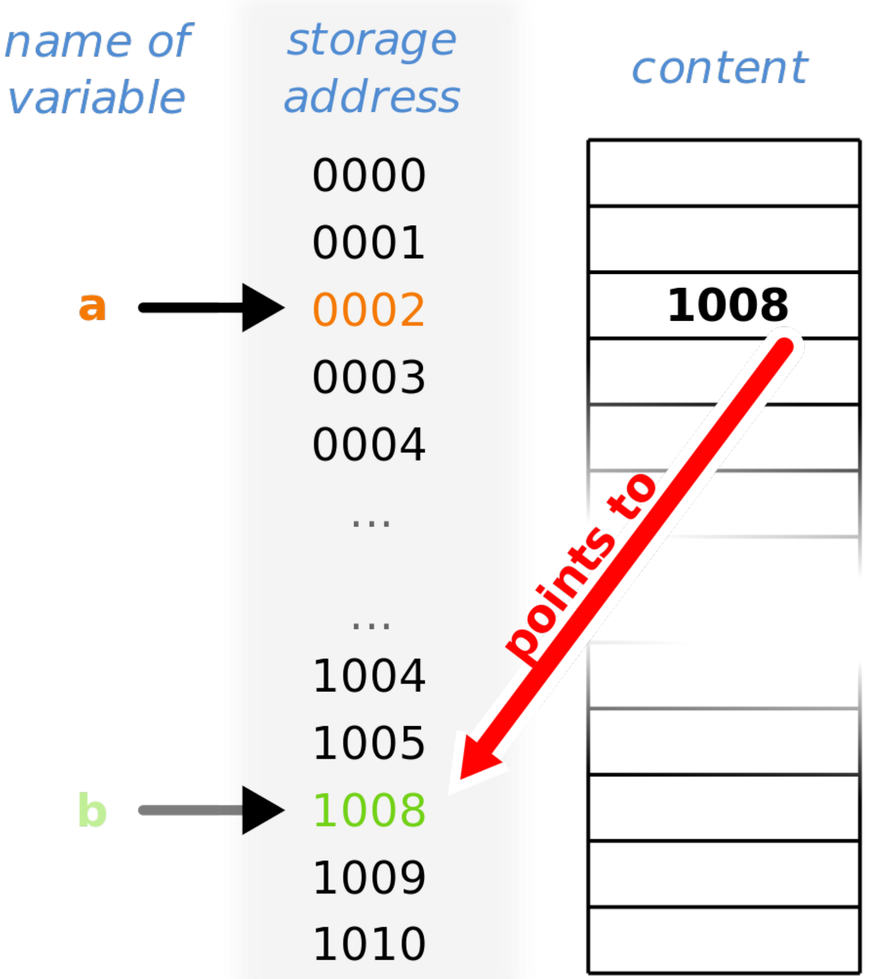
Reference types

Languages with references usually define a "**canonical**" pointer that inhabits the reference type (the same, for any associated type): **null**. Pointers should assume **null** until they are assigned. The usual way to initialise a pointer is to use a construct that allocates a value and returns a reference to that object, e.g., in C

```
int* p;
p = NULL;
p = malloc (sizeof (int));
```

C does not specify an implicit initialisation, so it is not safe to assume that `p` is valid after its declaration. This is why it is usually suggested to explicitly initialise pointers to `NULL` if we foresee to allocate their memory later in the program.

Then, we use `malloc` to allocate a specific amount of bytes (the `sizeof (int)`, above) on the heap. Since `malloc` ignores types, it always returns a `void` pointer (`*void`), indicating that it refers to a region of memory whose type is unknown.



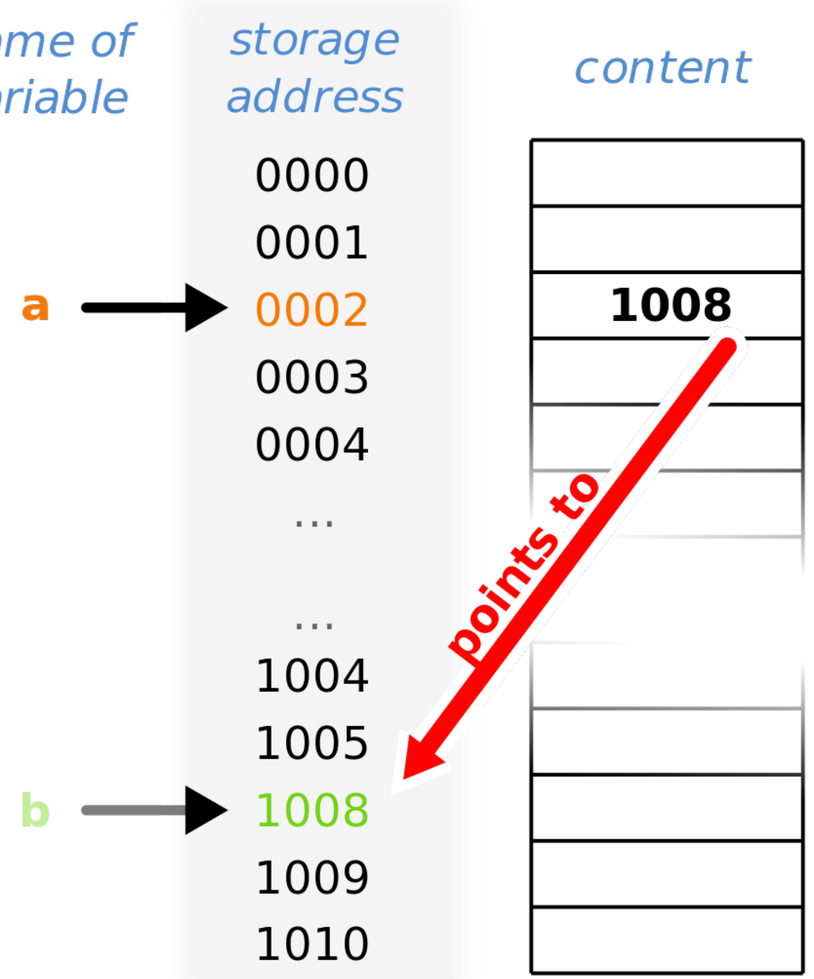
Reference types

Languages with references usually provide some **variable referencing operator**, i.e., to refer the memory location of variables. E.g., `&` in C

```
float pi = 3.1415;
float* p = NULL;
p = &pi;
```

The pointer `p` points to the location that contains the variable `pi`.

Contrarily to `malloc`, variable referencing lets pointers refer locations on the stack.



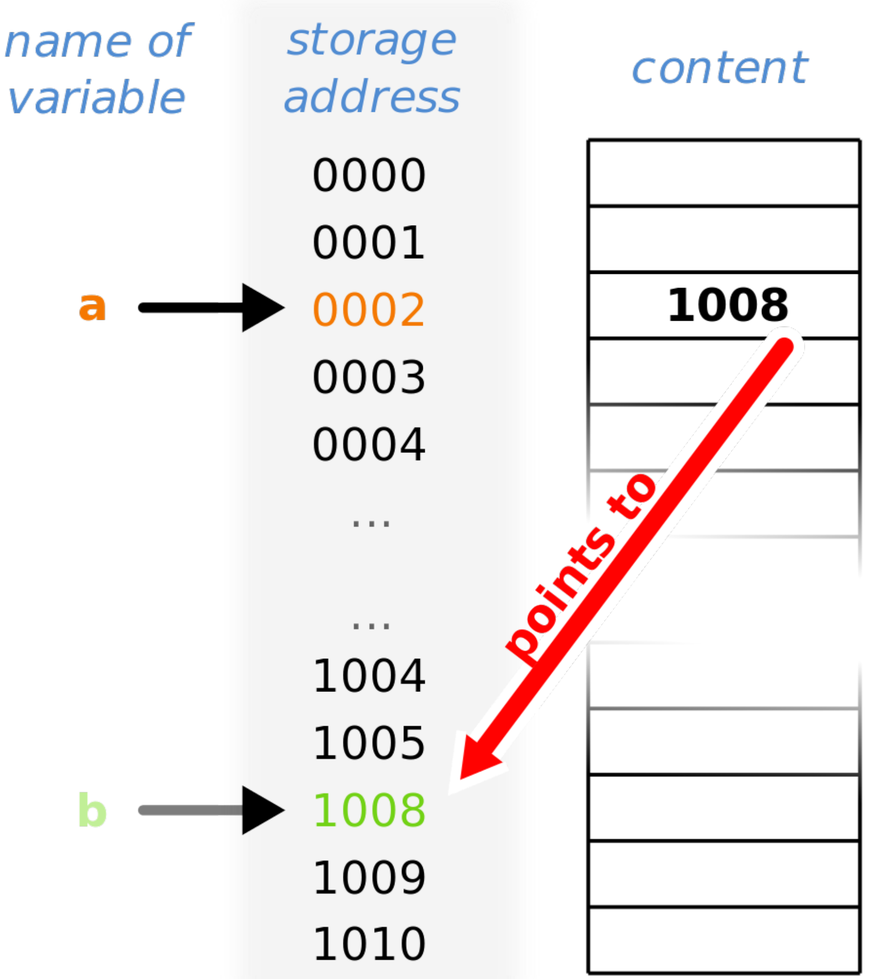
Reference types

Language with references usually also provide a **dereferencing operator**, e.g., `*` in C

```
float pi = 3.1415;
float* p = NULL;
p = &pi;
*p = *p + 1;
```

where we assign the value `4.1415` to `pi` by dereferencing `p` both in the left and right side of the assignment. The dereferencing on the left let us *read* the content of the location referenced by `p` (the content of the variable `pi`), while the dereferencing on the right let us *write* on the location referenced by `p` (the one corresponding to `pi`).

Note that the assignment does not modify the value of `p`, since it is always used in its dereferenced form.



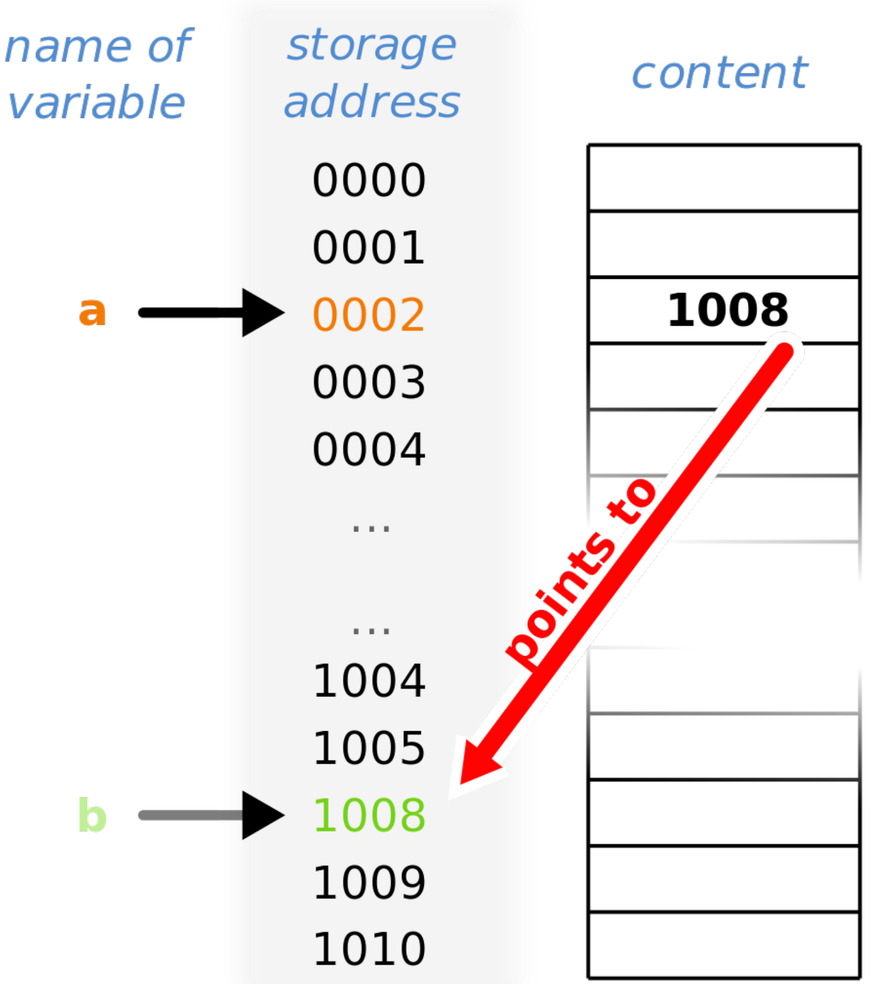
Reference types

Language with references can incur in **implicit deallocation**, e.g., (in C)

```
int* p = malloc (sizeof (int));
*p = 5;
p = null;
```

we can create an unaccessible memory region, since we destroyed the only pointer to reach it.

This “unreleased” pieces of memory can grow over time (as long as the program runs) and can incur in a phenomenon called “memory leak”. The problem of recovering these portions of memory has been subject to studies in different directions, from garbage collectors (e.g., Java) to type systems that prevent this kind of behaviours (e.g., Rust).

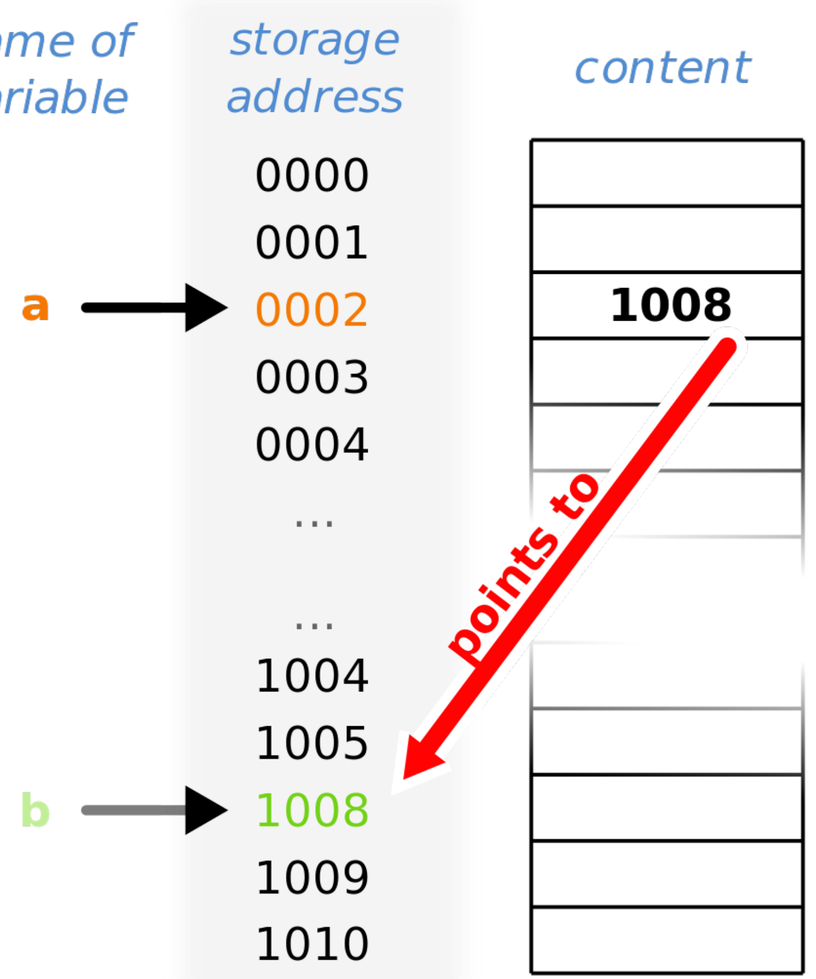


Reference types

Languages with references usually provide an **explicit deallocation operator** to release the memory referenced by a pointer. C provides the `free` operator

```
int* p = malloc (sizeof (int));
*p = 5;
free( p );
p = NULL;
```

As in the case of uninitialised pointers, it is a good practice to NULLify a freed pointer. Calling `free` on a pointer to the stack is a semantic error (it could lead to unpredictable behaviour).



Reference types, Rust

Rust is known for its safe treatment of pointers. The language provides the same operators as C, but puts in place static checks that allow the compiler to automatically free unused memory and prevent null references, dangling pointers, double frees, and pointer invalidation.

```rust
fn main() {
    let v: [i32;2] = [ 10, 11 ];
    let vp: *const [i32;2] = &v;
    unsafe {
      println!("{:?}", *vp);
    }
    println!("{:?}", *vp );
}
```

```
> rustc -o main main.rs
error[E0133]: dereference of raw pointer is unsafe and requires unsafe
 function or block
 --> main.rs:7:22
  |
7 |     println!("{:?}", *vp );
  |                       ^^^ dereference of raw pointer
  |
  = note: raw pointers may be NULL, dangling or unaligned; they can vi
olate aliasing rules and cause data races: all of these are undefined
behavior

error: aborting due to previous error

For more information about this error, try `rustc --explain E0133`.
exit status 1
>
```

`&T`	Allow one or more references to read T
`&mut T`	Allows a single reference to read and write T
`*const T`	Unsafe read access to T
`*mut T`	Unsafe read and write access to T
`Box<T>`	Heap-allocated T with a single owner that may read and write T
`Rc<T>`	Heap-allocated T with many readers
`Arc<T>`	Same as a Rc<T> but with safety guarantees for multi-threading

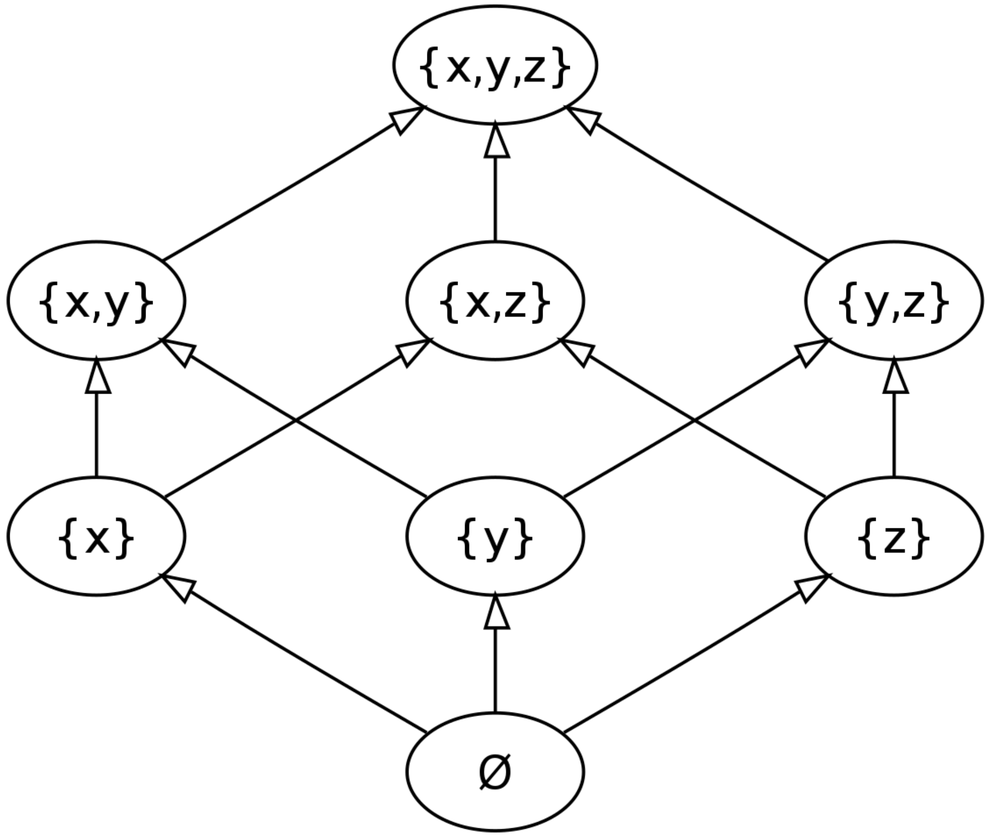
Power sets (towards product types)

We assume the axiom of power set, i.e.,

$$\forall S \; \exists P \; \forall R \; [\; R \in P \iff \forall Q \; (Q \in R \implies Q \in S) \;]$$

We call the set P from the axiom above the **power set** of S, also written $\wp(S)$. The right-to-left direction of $\iff$ implies that $\wp(S)$ is unique.

Informally, we can see $\wp(S)$ as the set of all the subsets generated by any combination (also the empty one) of elements in S.



Ordered Pairs and Cartesian Products (towards product types)

Let $a \in A$ and $b \in B$ and let $\big\{\{a\},\{a,b\}\big\} \triangleq (a,b)$ then

$\{a\} \subseteq A$, $\{b\} \subseteq B$

$\{a\} \in \wp(A)$, $\{b\} \in \wp(B)$

$\{a,b\} \subseteq A \cup B$ and $\{a,b\} \in \wp(A \cup B)$

$\big\{\{a\},\{a,b\}\big\} \triangleq (a,b) \subset \wp(A \cup B)$

$(a,b) \in \wp(\ \wp(A \cup B)\ )$

Let $\{(a,b) \mid a \in A \wedge b \in B\} \triangleq S \times T$

called the Cartesian **product** of S and T

$(a,b) \in A \times B \subset \wp(\ \wp(S \cup T)\ )$

Product Types

Arrays, sets, and pointers are examples of composite types that “take as parameter” one type.

When we combine two or more types in some fixed structure, we talk about **product types**.

The name comes from the notion of “direct product” from mathematics, which is a generalisation of the Cartesian product $A \times B \triangleq \{(a, b) \mid a \in A \wedge b \in B\}$

The most common product types are **pairs**, **tuples**, **records**, and **variants**.

By convention, the empty product is the Unit.

Product Types • Pairs and Tuples

The simplest form of product type is the **pair**. Given two types A and B, the pair type $A \times B$ denotes the set of all possible pairs (all the possible combinations) of the values in A and B. The generalisation of pairs are **tuples**, which define the product of an arbitrary, finite number of types $T_1, \cdots, T_n$ as $\prod_{i=1}^{n} T_i = T_1 \times \cdots \times T_n$

C and Java do not support directly pairs/tuples (one can implement them using records). Rust supports tuples with the syntax `(T1, …, Tn)`, e.g., `(i32, i32)` is a pair of integers that can represent a coordinate system. Since tuples define types based on the order of its components, Rust follows that abstraction to access the components of a tuple value, e.g., `coords.0`

```
let coords: (i32,i32);
coords=(89,97);
let x = coords.0;
let y = coords.1;
```

Product Types • Records

Records interpret type products by replacing the positional adjustment of type components in tuples with their identification by means of (distinct) labels.

Languages implement records in different ways, e.g., as **structures**, **classes** (Java 17 introduced records). The elements of records are usually called **fields**.

C

```c
struct Person {
 char name[ 5 ];
 int age;
};
```

```c
struct Person p = {
  .name = { 'E', 'v', 'a' },
  .age = 25
};
char* name = p.name;
int age = p.age;
```

Java

```java
record Person (
 char[] name,
 int age
){}
```

```java
Person p = new Person(
    new char[]{ 'e', 'v', 'a' },
    25
);
char[] name = c.name();
int age = c.age();
```

Rust

```rust
struct Person {
 name: [ char; 5 ],
 age: i32
}
```

```rust
let p: Person = Person {
  name: [ 'E', 'v', 'a', '', '' ],
  age: 25
};
let name = c.name;
let age = c.age;
```

Product Types • Records

In records, the order of fields is generally significant and possibly followed in their memory representation, e.g., storing fields in contiguous locations, even if bitwise alignment may entail gaps between fields (e.g., in `Person`, `names` will always take 5 bytes). While in `C` this is not evident, Rust forces us to always "fill" the possible, missing value with some default ones (e.g., the empty char values for "Eva"). Since Java allocates objects in the heap, the problem does not present itself (indeed, we cannot specify, in the type, constraints on the size of `name`).

C

```c
struct Person {
 char name[ 5 ];
 int age;
};
```

```c
struct Person p = {
  .name = { 'E', 'v', 'a' },
  .age = 25
};
char* name = p.name;
int age = p.age;
```

Java

```java
record Person (
 char[] name,
 int age
){}
```

```java
Person p = new Person(
    new char[]{ 'e', 'v', 'a' },
    25
);
char[] name = c.name();
int age = c.age();
```

Rust

```rust
struct Person {
 name: [ char; 5 ],
 age: i32
}
```

```rust
let p: Person = Person {
  name: [ 'E', 'v', 'a', '', '' ],
  age: 25
};
let name = c.name;
let age = c.age;
```

Product Types • Pattern Matching

While product types produce new data types, there is a powerful programming construct that helps consuming them in a structured way: **pattern matching**.

Pattern matching checks and locates specific elements against some pattern, e.g., in Rust

```
let x: i32 = 2;
let isEven = match x%2 {
  1 => true,
  _ => false
}
```

Safe implementations of pattern matching guarantee exhaustive matching, which help excluding common errors such as missing cases, impossible case, and redundant cases.

Product Types • Pattern Matching

While product types produce new data types, there is a powerful programming construct that helps consuming them in a structured way: **pattern matching**. Pattern matching checks and locates specific elements against some pattern, e.g., in Rust

```
struct Person { name: [ char; 3 ], age: i32 }
struct PersonR { name: [ char; 3 ], age: [ char; 4 ] }
let eva = Person{ name: ['E','v','a'], age: 25 };
let Person{ name, age } = eva;
let evaR = PersonR{ name, age: match age {
    1..=10 => [ 'K','i','d','!' ],
    11..=20 => [ 'T','e','e','n' ],
    _ => [ 'O', 'l', 'd', '!' ]
}}
```

Product Types • Recursive Types

Recursive types (as a concept) are useful to define data structures, such as Lists and Trees, that can dynamically grow. Product types are one way to express **recursive** types. Note, in the example below, in Java, the usage of **null** to “close” the structure.

```
record IntList( int n, IntList cons ){}

IntList l = new IntList(
  1, new IntList(
    2, new IntList( 3, null )
  )
);
```

Sum Types

Product types describe compositions of types. There might be cases where we want to denote that **some variable can hold a disjoint union of types**, e.g.,

$$\text{int} = \{-13, 0, 1, 17, \cdots\} \quad \text{int}^* = \{(-13, i), (0, i), (1, i), (17, i), \cdots\}$$

$$\text{char} = \{Y, 1, Z, 0, H, \cdots\} \quad \text{char}^* = \{(Y, c), (1, c), (Z, c), (0, c), (H, c), \cdots\}$$

$$\text{int} \sqcup \text{char} = \text{int}^* \cup \text{char}^* = \{(-13, i), (Y, c), (0, i), (1, i), (1, c), (17, i), (Z, c), (0, c), (H, c), \cdots\}$$

So that, declaring that x is of type int $\sqcup$ char means that x can either contain an integer or a char. The union of the **tagged sets** ($\text{int}^* \cup \text{char}^*$) tells us that, even if there might be coinciding elements in the sets, we always know to what set those values originally belong in (e.g., $(0, c)$ and $(0, i)$).

Sum Types

Disjoint unions of types are usually called **sum types** (but also **tagged unions**, **union types**, **choice type**, **variant types**, and **coproducts**).

A practical example of a sum type is that of an `Address` type, able to range over both `PhysicalAddress` and `VirtualAddress` types (e.g., a person's postal address and their email).

Some languages (especially, those inspired by Pascal and of the ML family) provide direct support to sum types via some dedicated operator, e.g.,

```
type Address = PhysicalAddress + VirtualAddress
```

Sum Types

Besides having explicit operators in the language, we already saw a type that can help up define sets of values: `enumerations`. Languages like Java and Rust extended `enumerations` to capture the case of sum types. E.g.,

```
enum Address {
    PhysicalAddress { long: i32, lat: i32 },
    VirtualAddress { email: [ char; 20 ] }
}
let a = Address::PhysicalAddress{ long: 15, lat: 25 };
match a {
  Address::PhysicalAddress{ long, lat } => …,
  Address::VirtualAddress{ email } => …
}
```

Sum Types

Besides using `enumerations` (as in Rust), Java recently introduced `sealed` classes, which define the only data structures `permitted` to appear as one of the possible types present in a given sum type.

```
sealed class Address permits
Address.PhysicalAddress, Address.VirtualAddress {
 static class PhysicalAddress extends Address {
  int lon; int lat;
  PhysicalAddress( int lon, int lat ) { … }
 }
 static class VirtualAddress extends Address {
  char[] email;
  VirtualAddress( char[] email ) { … }
 }
}
Address a = new Address.PhysicalAddress( 15, 25 );
switch ( a ) {
 case Address.PhysicalAddress p -> …;
 case Address.VirtualAddress v -> …;
 default -> …
}
```

Sum Types

C has a notion of union of structures. C unions are a way to have the same memory location hold different types of data—e.g., an integer or a char—, where the memory is allocated according to the biggest structure—e.g., that the size of char, in the previous example.

However, the language does not discipline the way in which users interact with (the location of a) union variable—e.g., given a variable x of the union type in the example above, we can write an integer in it and then read it as a char, without any error/warnings raised by the compiler.

```
union Data {
    int i;
    char c;
};

union Data data;
data.i = 10;
data.c = 'A';

// data.i = 65
```

Sum Types • Recursive Types

Sums are an alternative to product types (no need for **null**s) for recursive types.

```
sealed class IntList permits
  IntList.Cons, IntList.End {
   static class Cons extends IntList {
    int n; IntList cons;
    Cons( int n, IntList cons ){…}
   }
   static class End extends IntList {
    End(){}
   }
}
```

```
IntList l = new IntList.Cons( 1,
 new IntList.Cons( 2,
  new IntList.Cons( 3, new IntList.End() ) ) );
```

Relations (towards Functions)

Given a sequence of sets $S_1, \ldots, S_n$, we call the set $\mathbb{R} \subseteq S_1 \times \cdots \times S_n$ a **relation** on the Cartesian product $S_1 \times \cdots \times S_n$ when $\mathbb{R}$ relates the elements $s_1 \in S_1, \cdots, s_n \in S_n$, i.e., when, for some $s_1, \cdots, s_n$, $(s_1, \cdots, s_n) \in \mathbb{R}$.

When $\mathbb{R} \subseteq S \times T$ we say that $\mathbb{R}$ is a **binary relation**. Given $s \in S$ and $t \in T$ if $(s, t) \in \mathbb{R}$ we usually also write $s \ \mathbb{R} \ t$. Conventionally, with $\mathbb{R} \subseteq S \times T$, we call the elements of S in $\mathbb{R}$ the **domain** of $\mathbb{R}$ ($\text{dom}(\mathbb{R}) \subseteq S$) and the elements of T in $\mathbb{R}$ the **range** of $\mathbb{R}$ ($\text{ran}(\mathbb{R}) \subseteq T$)

Functions

When $\mathbb{R} \subseteq S \times T$, $s\ \mathbb{R}\ t$, $s\ \mathbb{R}\ t'$, and $t = t'$ then we call $\mathbb{R}$ a **partial function**.

When $\mathbb{R}$ is a partial function, we usually adopt the notation $\mathbb{R}(s) = t$ (alternative to $(s, t) \in \mathbb{R}$ and $s\ \mathbb{R}\ t$) and we call s the **argument** of $\mathbb{R}$ and t the **value** of $\mathbb{R}$ for s. We also say that $\mathbb{R}$ **maps** s into t and we adopt an alternative (mapping) notation $\mathbb{R} \subseteq S \times T \equiv \mathbb{R} : S \to T$

When $\text{dom}(\mathbb{R}) = S$ we can $\mathbb{R}$ a **total function**. Unless specified differently, when talking about functions, we intend total ones.

Functions

Given $f : S \to T$, $f \subset \wp(\ \wp(S \cup T)\ )$ and $f \in \wp(\ \wp(\ \wp(S \cup T)\ )\ )$, by definition $\wp(S \times T) \triangleq T^S$, then $f \in T^S$

Given the definition above, we have an alternative way of writing $\wp(S)$ as 2^s.

We can do this because we consider the **characteristic function** χ_Q of a subset Q of a set S such that $\chi_Q : S \to 2$ (with $2 \triangleq \{0,1\}$) where $\chi_Q(q) = 1$ when $q \in Q \cap S$ and 0 otherwise.

χ induces a family of functions each describing one subset Q of S, i.e., we have a function that defines a one-to-one correspondence (bijection) between each element in $\wp(S)$ and its characteristic function in 2^S.

Function types

High-level languages frequently support the definition of functions (or procedures), however, only a few denote the type of functions (i.e., give them a name in the language). E.g., if `f` is a function defined as `R f( P p ){ … }`, we can denote its type as `P->R`, where `P` is the type of the unary parameter accepted by `f` and `R` is the type of the value returned by f. The set-theoretic representation of `P->R` is R^P.

This naming discipline follows the polyadicity of functions, e.g., a function of shape

`R f( P1 p1, …, Pn pn ){ … }` has type `P1->…->Pn->R` or $R^{P1^{\cdots^{Pn}}}$

The values of a function type are denotable in all languages, but only some (so called, “functional” languages) make them expressible (or storable). The main operation allowed on a function type value is the **application**, i.e., the invocation of a function on some arguments (actual parameters).

The Algebra of Types

Product, Sum, and Function types recall the existence of some algebra—a discipline that defines the rules for manipulating (type) symbols—of types, therefore defined "**algebraic** (data) **types**".

Type systems can make use of the properties of this algebra to express and check properties of programs.

Types	Algebra	Inhabitants
`Void`	0	the empty type/symbol
`Unit`	1	the singleton-value type
`Bool, Char`	n	
`A + B`	a + b 0 + a = a + 0 = a	The sum of the inhabitants of A and B
`A x B`	a x b a x b x 1 = a x b	The product of all inhabitants of A and B
`A -> B`	b^a	The combinations of B given A, e.g., Unit -> Bool (2^1) and Bool -> Unit (1^2)

Bool -> Unit		(A) Bool	
		true	false
(B) Unit	f1	unit	unit

Unit -> Bool		(A) Unit
		unit
(B) Bool	f1	true
	f2	false

Bool -> Bool		(A) Bool	
		true	false
(B) Bool	f1	true	true
	f2	true	false
	f3	false	true
	f4	false	false

inspired to Burget, Joel. "**The Algebra (and Calculus!) of Algebraic Data Types.**" *Codewords.recurse.com*.

Type equivalence

One of the main questions we can ask about types of a program is

“when are two types equal?”

which underpins some of the correctness tests of type checking.

Answering questions on equality does not have a single interpretation, as it might depend on the context from where we are checking equality.

For example, let `P` be a type defined as a subset of integers and `f` a function that can sum any two `integers`. From the perspective of `f`, we can consider `P` as equivalent to integers, since we know the function can work on a superset of the values in `T`. Conversely, if `f` accepted only values of `T`, we cannot safely assume integer parameters as equal to `T` as, e.g., the body of the function might consider some invariant from `T`, invalidated by the `integers`.

Preorders and equivalences

Given a set S, we call the set $\mathbb{R}$ a binary relation on S when $\mathbb{R}$ is a subset of the Cartesian product of S by itself, i.e., $\mathbb{R} \subseteq S \times S$. Given two elements $\{s_1, s_2\} \subseteq S$ we say they are in the relation, denoted $s_1 \ \mathbb{R} \ s_2$, if $(s_1, s_2) \in \mathbb{R}$. Binary relations have different of properties:

- **Reflexive**: for any s, $(s, s) \in \mathbb{R}$;
- **Symmetric**: for any s_1 and s_2, $\{(s_1, s_2), (s_2, s_1)\} \subseteq \mathbb{R}$;
- **Antisymmetric**: for any s_1 and s_2 $\{(s_1, s_2), (s_2, s_1)\} \subseteq \mathbb{R}$ implies $s_1 = s_2$;
- **Transitive**: for any s_1, s_2, and s_3 $\{(s_1, s_2), (s_2, s_3)\} \subseteq \mathbb{R}$ implies $(s_1, s_3) \in \mathbb{R}$

When $\mathbb{R}$ is reflexive and transitive, we call $\mathbb{R}$ a **preorder**. When $\mathbb{R}$ is a symmetric preorder, we call it an **equivalence**. When $\mathbb{R}$ is an antisymmetric preorder we call it a **partial order**.

Nominal type equivalence

In type systems that consider a nominal notion of type equivalence each new type definition introduces a new name, different from any existing one.

Let name$(T) = n$ be a function that, given a type T, it gives us its associated name n, then $T_1 \; NTE \; T_2 \iff$ name$(T_1) =$ name(T_2).

Hence although the types `Dollar = int` and `Euro = int` are functionally indistinguishable to e.g., a function that takes one or the other as parameter, in a nominal system they are not equivalent. Although quite simple, the idea behind nominal system is the one most adherent to the programmers intention: e.g., if the programmer used types to distinguish between Dollars and Euros, there might be some invariants (e.g., their denominations) not captured a the level of types that the programmer rely upon in the body of functions.

Duck Typing

While nominal type-checking is usually performed statically, we know that type-checks can also happen at runtime (the main case being that of interpreted languages).

A popular way of performing type checking at runtime is via the so called **duck typing** method, which works by checking if a given value supports the operators expected by the program.

```
sum( p ){ return p.x + p.y }
loc( p ){ return p.x % p.y % p.z }
c = { x: 15, y: 25, z: 63 }
s = { x: 64 , y: 17 }
sum( c ) // 40
sum( s ) // 81
loc( c ) // 15
loc( s ) // Error: s has no field 'z'
```

Structural Type equivalence

Duck typing introduced an alternative to nominal type equivalence: as long as we cannot observe structural differences between values, we can consider them of the same type.

In general, this interpretation takes the name of **structural type equivalence** and we can also perform it statically. However, since we do not know in advance what paths values will take in the program, we need to perform more conservative checks than the “operational” ones seen for duck typing: we test types for equivalence by comparing all their operations, structures, and subelements.

Of course, this makes the definition of structural equivalence more involved, e.g.,

$$(\mathsf{T}_{a1}, \cdots, \mathsf{T}_{an})\ STE\ (\mathsf{T}_{b1}, \cdots, \mathsf{T}_{bn}) \iff \forall i \in [1,n],\ T_{ai}\ STE\ T_{bi}$$

$$\text{struct } \mathsf{T}_a\{f_1 : \mathsf{T}_{a1}, \cdots, f_n : \mathsf{T}_{an}\}\ STE\ \text{struct } \mathsf{T}_b\{f_1 : \mathsf{T}_{b1}, \cdots, f_n : \mathsf{T}_{bn}\} \iff \forall i \in [1,n],\ T_{ai}\ STE\ T_{bi}$$

On the adoption of Nominal vs Structural type systems

Besides embodying a tight notion of equivalence, nominal typing has several other benefits:

- a direct link for the runtime to e.g., print, marshal, and check type coercion (spoiler);
- an intuitive denotation of recursive types—types whose definition refers the type itself like lists and trees (e.g., a `List` of `Lists` or even mutually recursive ones, e.g., `Trees` of `Lists` of `Trees`); $\mathsf{IntList} := (\mathsf{int} \times \mathsf{IntList}) + \mathsf{Unit} \quad \mathsf{ListIntList} := (\mathsf{ListInt} \times \mathsf{ListIntList}) + \mathsf{Unit}$ *vs* $\mu t\,.\,((\mu t'\,.\,(\mathsf{int} \times t') + \mathsf{Unit}) \times t) + \mathsf{Unit}$
- checking subtyping (spoiler) is a(n almost trivial) direct check of the nominal, declared subtype relations among the named types.

These advantages decreed the “success” of nominal type systems, present in many mainstream programming languages, e.g., Java and Rust. C also has a prominently nominal type system, although the `typedef` declaration allows users to equate different types with coinciding aliases (this feature is usually regarded as being unsafe, e.g., recall the `Dollars` vs `Euros` example).

Type compatibility

The example for duck typing showed that we can correctly use a structure containing the fields x, y, and z in a function `sum` that just asks for values with the x and y fields. This means that sometimes we can use a weakened version of equivalence and still obtain correct program. This weakened form of equivalence is usually called **type compatibility**.

Formally, since equivalence is a symmetric preorder, it subsumes compatibility, which is a preorder (reflexive and transitive), but not the other way around (not all compatible types are equivalent).

Type compatibility

The compatibility relation varies among languages. Some common interpretations of compatibility (besides equivalence) are, let T and S be two types:

- the values of S are a subset of the values of T, e.g., intervals;
- the values of S are a subset of canonically-correspondent values of T. This is typical of types like, e.g., `float` and `int` types, where any `int n` has a canonical corresponded `float n.0`;
- the values of S are a subset of arbitrary-correspondent values of T. Here, we drop the requirement of canonicity from the previous point and assume the presence of some arbitrary transformation that converts any value in S to a value of T, e.g., we can make `int` and `float` compatible by converting `floats` (e.g., via rounding) into `ints`;
- all operations on the values of T are also possible on the values of S. This is the example shown for duck typing and the principle behind some notion of subtyping (spoiler);

Type coercion and type casting

The last three points on alternative notions of type compatibility assume the existence of some **type conversion** mechanism, able to bridge the differences between values of different types. Also in this case, languages adopt two ways (frequently mixed) of performing these conversions:

We call **type coercion** the implicit application of some canonical/arbitrary type conversion. An example is e.g., a sum function that accepts `floats` and, if we pass to it `integers`, the compiler/interpreter inserts the necessary conversions implicitly, without reporting a compatibility error. In both cases, conversions are either **syntactic**, when the types share the same representation in memory (this is the case, e.g., of intervals, where no conversion applies) or happen via some canonical/arbitrary **conversion**, which transforms the memory representation of a value of some type into a value of another—e.g., integers into floats (canonical) and vice versa (arbitrary).

We call **type casting** the explicit annotation in the language of a type (and value) conversion, which applies some user-defined conversion procedure. Type casting has also a documentation value, making type conversions statically explicit. As an example of type casting, C and Java adopt a minimalistic syntax `S s = ( S ) t` while Rust provide a more verbose one `let s: S = t as S`.

Type inference

The type checker of a language verifies that a program respects the rules imposed by the type system (in particular, compatibility). To perform its checks, the type checker must determine the type of the expressions present in the program, using the information on types that the programmer has inserted in the program.

Concretely, the type checker determines the type of expressions by visiting the parse tree of the program, starting from its leaves (variables and constants whose type is known), it descend to the root and **calculates the type of the expressions from the information accumulated along its path** (e.g., the type system could establish that `+` is an operator which, applied to two expressions of type `int` results into an expression of type `int`).

Knowing that the type checker can **infer** some information from a reduced amount of type annotations, languages can spare the programmer the task of annotating all expressions. **Type inference** is the process of attributing types to expressions, omitting explicit type annotations.

Type inference

Sometimes, the inference algorithm cannot directly infer the type of some expression, but it rather needs to keep its type “open”, proceed with other parts of the program, and “return” on that expression later on—of course, if it collected all the available information and it cannot still fix the type of the considered expression, the algorithm “gives up” and report to the user the need for more information.

Technically, keeping the type of the expression “open” means assigns to the latter a *type variable*, which, proceeding with the exploration of the parse trees, it enriches with constraints (e.g., we might meet a + operation applied on it, which limits the range of possible types to only those that support it). The procedure that performs this check on the constraints is a renowned resolution strategy from logic programming known as the **unification** algorithm.

C does not provide relevant support for type inference. Java and Rust provide simple forms of it. Languages of the ML family, based on the Hindley-Milner type system [1,2], provide more complete type inference support.

[1] Hindley, J. Roger (1969). "**The Principal Type-Scheme of an Object in Combinatory Logic**". Transactions of the American Mathematical Society. 146: 29–6
[2] Milner, Robin (1978). "**A Theory of Type Polymorphism in Programming**". Journal of Computer and System Sciences. 17 (3): 348–374.