Tipi Base e l'Algebra dei Tipi

Sistemi di Tipi

Ogni linguaggio di programmazione ha un proprio **sistema di tipi**, cioè le informazioni e le regole che governano i tipi e i loro valori, chiamati **abitanti** (inhabitants) del tipo. Più precisamente, un sistema di tipi comprende

1. un insieme di **tipi di base**;
2. meccanismi per **definire nuovi tipi**;
3. meccanismi di **computazione** sui tipi, che includono:
    1. **regole di equivalenza**, che specificano quando due tipi corrispondono allo stesso tipo;
    2. **regole di compatibilità**, che specificano quando si può usare un tipo al posto di un altro;
    3. **regole/tecniche di inferenza**, che specificano come assegnare un tipo a un'espressione, a partire dalle informazioni sui suoi componenti;
4. la definizione sul controllo dei vincoli di tipo statico o dinamico.

Tipi Base

I tipi base (primitivi/semplici/scalari) sono tutti i tipi che definiscono i valori denotabili del linguaggio. Valori denotabili comuni sono `42`, `3,14` e `'A'` e i loro tipi (di base) possono essere chiamati rispettivamente `int`, `float` e `char`. Usiamo il verbo “possono” perché i valori, e quindi i loro tipi, non hanno la stessa definizione in tutti i linguaggi.

Il modo in cui caratterizziamo `42` e `int` dipende dal linguaggio che stiamo considerando. In Java, useremmo 4 byte per rappresentare `42` e il tipo `int` comprenderebbe tutti i numeri interi da -2^{31} a $2^{31}-1$. In Rust, dovremmo specificare ulteriormente il “tipo” di intero che ci interessa memorizzare, ad esempio il tipo `i32` ci darebbe una dimensione e un intervallo simile a quello di Java, mentre il tipo `u32` prenderebbe in considerazione solo i numeri positivi (quindi l'intervallo da 0 a $2^{32}-1$).

Poiché i linguaggi adottano sintassi diverse, forniscono anche modi diversi per dichiarare i tipi di base.

Java adotta la sintassi `nomeTipo nomeVariabile`

Rust fornisce la dichiarazione `let nomeVariabile : nomeTipo`

Tipo Unit (vs Void)

Il tipo più elementare è quello che contiene un solo elemento (l'insieme è un singoletto).

L'unico abitante del tipo `Unit` è l'unità singoletto (rappresentato anche come `()`) e di solito è associato a operazioni il cui tipo di ritorno non è utilizzabile (ad esempio, perché l'operazione ha generato solo effetti non tracciati dal sistema di tipi (side effects), come la stampa sullo schermo). Questo è coerente con una mappatura dell'input (qualunque esso sia) verso lo stesso, imperscrutabile output (l'unità).

Linguaggi come Java e C hanno un concetto simile all'unità col tipo `void` che, tuttavia, presenta alcune differenze con `Unit`. Per esempio, mentre possiamo passare l'unità come argomento delle operazioni, non possiamo ottenere né passare un `void`: come dice il nome, rappresenta il "vuoto" e non si possono definire altri tipi usando sottocomponenti di `void`. Questa discrepanza con Unit diventa visibile, ad esempio, nei generici di Java, che hanno richiesto l'introduzione del tipo `Void` di cui il riferimento nullo (`null`) è l'unico abitante.

Tipi Booleani

I booleani indicano il tipo di valori logici e di solito comprendono:

- valori: i due valori di verità, `true` e `false`;
- operazioni: le principali operazioni logiche, come la congiunzione (&), la disgiunzione (|), la negazione (!), l'uguaglianza (==), l'or esclusivo (^), ecc.

Quando è presente (ad esempio, ANSI C non ha un tipo di questo tipo), i suoi valori sono denotabili, esprimibili e memorizzabili. È interessante notare che, mentre si suppone che un bit sia sufficiente per memorizzare i booleani, la rappresentazione effettiva in memoria dipende dal modello hardware del linguaggio, dall'unità indirizzabile di base dell'architettura e da altri requisiti di allineamento.

Ad esempio, in Rust le variabili di tipo `bool` richiedono un byte. Nella Java Virtual Machine i bool richiedono 2 byte (8 come intestazione, 1 per il valore e 7 di padding).

Tipi Carattere

I caratteri (characters o chars) indicano tutti i caratteri di un determinato (e definito a livello linguistico) insieme di simboli e di solito includono:

- valori: un insieme di codici di caratteri, ad esempio due insiemi comuni sono ASCII e UNICODE;
- operazioni: altamente dipendenti dal linguaggio; di solito troviamo uguaglianza (==), confronti (<,>).

I valori sono denotabili, esprimibili e memorizzabili e, di solito, la rappresentazione in memoria consiste in 1 (ASCII) o 2 byte (UNICODE).

Tipi Interi

Gli interi denotano un intervallo di numeri interi e di solito includono:

- valori: un sottoinsieme finito di numeri interi, normalmente fissato al momento della definizione del linguaggio e, a seconda della dimensione del byte di memorizzazione per la rappresentazione , che va da $[-2^{r-1}, 2^{r-1} - 1]$ per quelli con segno (con rappresentazione complemento a due) e $[0, 2^{r} - 1]$ per quelli senza segno. Alcuni linguaggi hanno un supporto integrato per interi di lunghezza arbitraria;

- operazioni: uguaglianza ($=$), confronti (<,>) e le principali operazioni aritmetiche (+, −, ∗, /, %).

I valori sono denotabili, esprimibili e memorizzabili.

Tipi Reali

I reali denotano un intervallo di numeri reali e di solito includono:

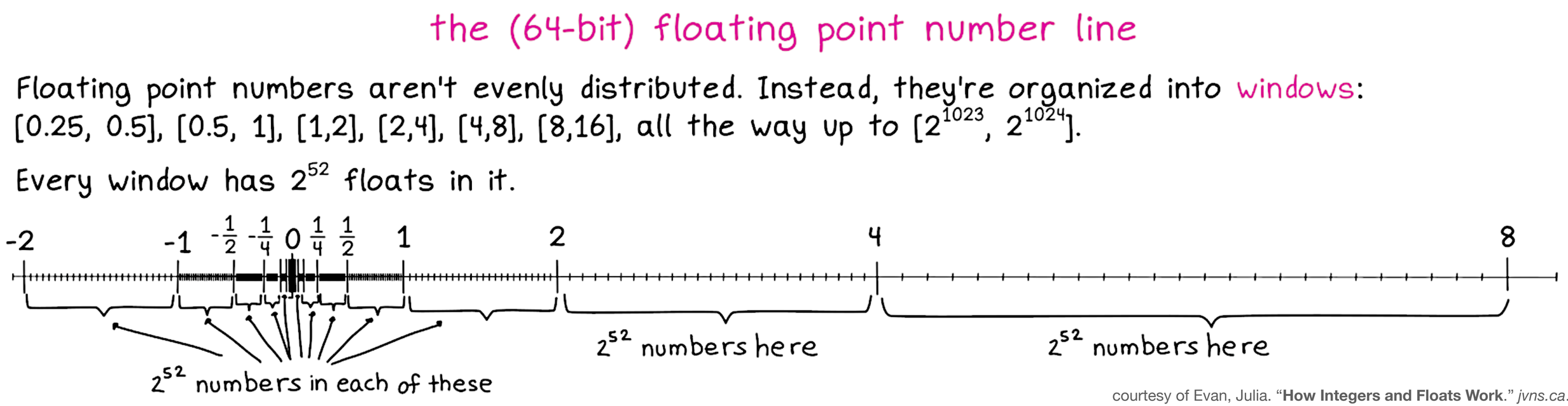
- valori: un sottoinsieme finito dei reali, normalmente fissato al momento della definizione del linguaggio. Vengono memorizzati principalmente tramite una rappresentazione a **virgola fissa** o a **virgola mobile**. Entrambi rappresentano i reali separando i loro numeri interi e decimali.
  - I numeri a virgola fissa riservano bit specifici per gli interi e i decimali. Utilizzando un formato con segno da n byte, con f su n bit per i decimali, abbiamo un intervallo $[-2^{n-1}/2^f, 2^{n-1}/2^f]$ con numeri a distanza costante di $1/2^f$.
  - I numeri in virgola mobile utilizzano il formato $s \cdot m \cdot b^e$, dove s è il segno (omesso se senza segno), m è il numero (mantissa), b la base e e è l'esponente che posiziona il float. Lo standard IEEE 754 definisce due formati, entrambi con $b = 2$, ma con precisione/esponente singolo (8 byte) e doppio (11 byte).
- operazioni: uguaglianza (==), confronti (<,>) e le principali operazioni aritmetiche (+, -, *, /, %).

I valori sono denotabili, esprimibili e memorizzabili.

Tipi Reali

I reali denotano un intervallo di numeri reali e di solito includono:

- valori: un sottoinsieme finito dei reali, normalmente fissato al momento della definizione del linguaggio. Vengono memorizzati principalmente tramite una rappresentazione a **virgola fissa** o a **virgola mobile**. Entrambi rappresentano i reali separando i loro numeri interi e decimali.



courtesy of Evan, Julia. "**How Integers and Floats Work**." *jvns.ca*.

Enumeration types

Un tipo di enumerazione consiste in un insieme finito di costanti, ciascuna caratterizzata da un proprio nome. C, Rust e Java (e altri linguaggi) forniscono la stessa sintassi, ad es,

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

che introduce un nuovo tipo chiamato RogueOne costituito da un insieme di 6 elementi, ciascuno contrassegnato dal proprio nome. Le operazioni disponibili sugli enum consistono in confronti e in un meccanismo per ottenere tutti i valori o passare da uno all'altro. Da un punto di vista pragmatico, gli enum presentano due vantaggi: 1) aiutano la leggibilità, poiché i nomi dei valori costituiscono una chiara forma di auto-documentazione del programma e 2) permettono al controllo di tipo di verificare che una variabile con enumerazione assuma solo i valori corretti.

Non tutti i linguaggi integrano gli enum in modo sicuro, ad esempio in C la scrittura sopra

è uno zucchero sintattico per

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

che equipara gli interi ai RogueOne e impedisce di distinguerli (e di verificarne la correttezza).

Extensional vs Intensional types

Gli interi (e i `float`, i `chars`, ...) rispetto agli `enum` hanno una differenza importante: l'utente specifica gli `enum` in modo **estensionale**, cioè elencando tutti i possibili abitanti di quel tipo. Al contrario, i linguaggi specificano gli interi, i float e così via in modo **intensionale**, cioè mediante predicati che definiscono la loro appartenenza ad alcuni domini di valori possibili (ad esempio, numeri interi a 32 bit, numeri in virgola mobile, caratteri UNICODE).

La logica tra l'utilizzo di definizioni intensionali o estensionali è:

- intensionali quando si dispone di un insieme definito di proprietà che identificano solo gli abitanti (valori validi) del tipo che stiamo definendo, con il vantaggio di risparmiare memoria se l'insieme degli abitanti è grande e di rendere possibile la definizione, nel caso di insiemi infiniti.
- estensionali quando è più efficiente (per spazio o computazione) specificare gli abitanti del tipo o non abbiamo un insieme chiaro di regole che li definiscono (ad esempio, un modo intensionale di definire il nostro tipo RogueOne potrebbe essere attraverso una regola del tipo "*i 6 personaggi principali del film RogueOne*" ... o no 😐?).

Tipi composti

I tipi enum alla C hanno introdotto in maniera surrettizia un nuovo concetto: possiamo creare nuovi tipi componendo quelli di base.

Nelle enumerazioni di C, abbiamo creato **insiemi di elementi denominati**, che corrispondono a numeri interi, ma sono possibili altre strutture, tra cui le più basilari sono: **array**, **insiemi** e **puntatori**.

Tipi Array

Un tipo **array** denota un insieme di elementi di un certo tipo, ciascuno indicizzato da almeno una **chiave identificativa** di un certo tipo (quando sono coinvolte 2 o più chiavi, si parla di array multidimensionali, ad esempio matrici, datacube, ecc.)

La nozione più comune di array assume le chiavi come numeri interi non negativi all'interno di un intervallo (di solito si considera l'intervallo $[0, n]$ per $n + 1$ elementi, che ne semplifica la disposizione in memoria) e lascia che sia l'utente a definire il tipo degli elementi. Altre forme di array, solitamente chiamate mappe o **array associativi**, permettono all'utente di fissare sia il tipo delle chiavi che quello degli elementi.

Tipi Array

Vediamo la sintassi di C, Java e Rust per dichiarare un array lineare di interi:

`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]`

Sia il C che Rust fissano nella dichiarazione del tipo la dimensione dell'array (3), mentre Java astrae da questo e lascia che sia l'inizializzazione a definire la dimensione dell'array (ne parleremo in seguito).

La maggior parte dei linguaggi (tra cui C, Java e Rust) estendono le dichiarazioni di array lineari a quelle a più chiavi

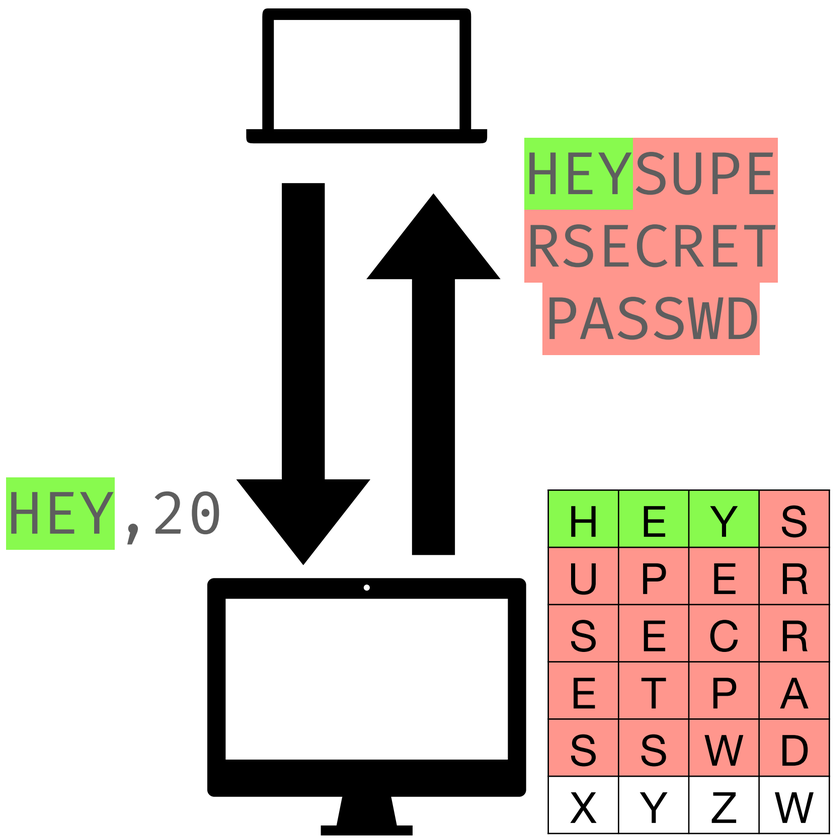
`int x[10][10]`	`int[][] x`	`let x: [[i32;10];10]`

Mentre C, Java e Rust uniscono il concetto di array multidimensionali e di array di array (in quest'ultimo caso), alcuni linguaggi (ad esempio, il Pascal) li tengono separati.

Tipi Array

L'operazione più semplice su un array è la **selezione** di un elemento tramite il suo indice. La notazione più comune (C, Java, Rust) è `a[e]` dove a è la variabile di tipo array ed e è un'espressione. Per gli array multidimensionali, le sintassi più comuni sono `a[e][e][e]` o `a[e,e,e]` — la seconda è per i linguaggi che hanno sia array multidimensionali che array di array. Altre operazioni su array interi sono, ad esempio, **l'assegnazione** (=), i **confronti** (==,<,>) e le **operazioni aritmetiche** (eseguite a coppie).

Poiché conoscono il tipo di indice degli array, i linguaggi safe **verificano che ogni accesso a un elemento dell'array avvenga davvero entro i suoi "limiti"** (poiché non ha senso accedere a elementi inesistenti). Tranne che in alcuni casi speciali, questo controllo può avvenire solo a tempo di esecuzione, ed è per questo che i linguaggi sicuri inseriscono controlli appropriati a ogni accesso. Linguaggi come Java e Rust garantiscono questa invariante a runtime (sollevando un errore/eccezione quando viene violata), ma il C non lo fa. Se da un lato questi controlli rallentano (un po') i programmi, dall'altro prevengono gli attacchi di **buffer-overflow**.

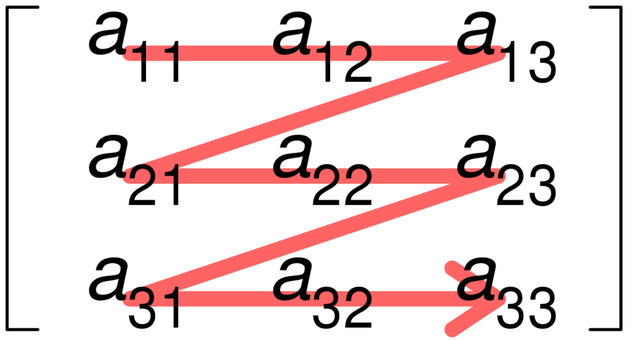


Tipi Array

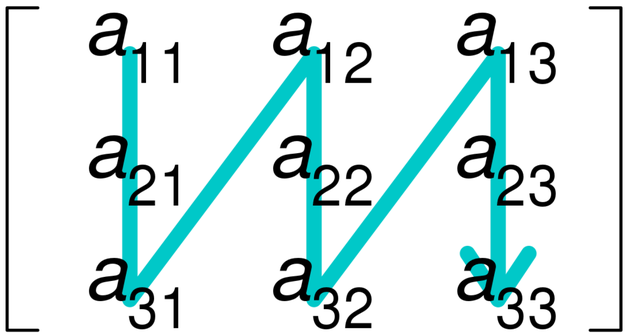
Un array viene solitamente memorizzato in una porzione contigua di memoria. Per un array monodimensionale, l'allocazione segue l'ordine degli indici. Per gli array multidimensionali, esistono principalmente due tecniche, chiamate **ordine di riga** (row major) e **ordine di colonna** (column major).

Nell'ordine di riga, due elementi sono contigui se differiscono di uno nell'ultimo indice. Nell'ordine per colonna, due elementi sono contigui se differiscono di uno nel primo indice. L'ordine di riga è un po' più comune di quello per colonna, soprattutto perché gli accessi per riga sono più frequenti di quelli per colonna. In generale, il principio di località del caricamento in funzione dei cache-miss **favorisce gli algoritmi che esplorano gli array per righe con ordine di riga e viceversa per l'ordine di colonna**.

Row-major order



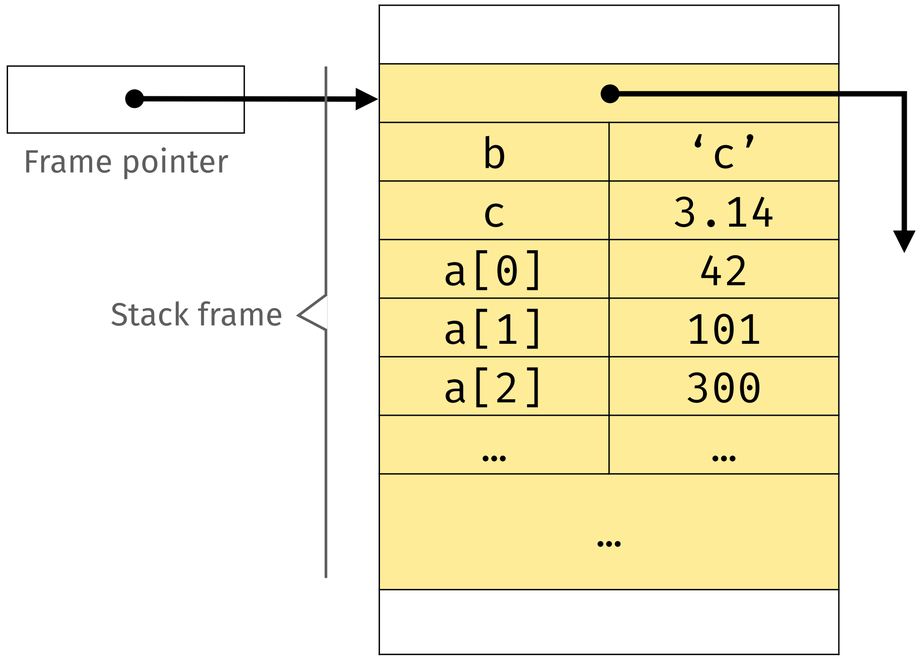
Column-major order



Tipi Array

Il **numero di dimensioni** e i **loro intervalli** determinano la forma di un array. Un aspetto importante della definizione di un linguaggio è decidere **se e quando fissare la forma degli array**. Se la forma è **fissa**, possiamo decidere di definirla in **fase di compilazione** (per i linguaggi compilati, come in C e Rust) o **quando elaboriamo la dichiarazione** (a tempo di esecuzione). In alternativa, possiamo avere **array dinamici**, la cui forma è determinata e cambia in fase di esecuzione (come in Java).

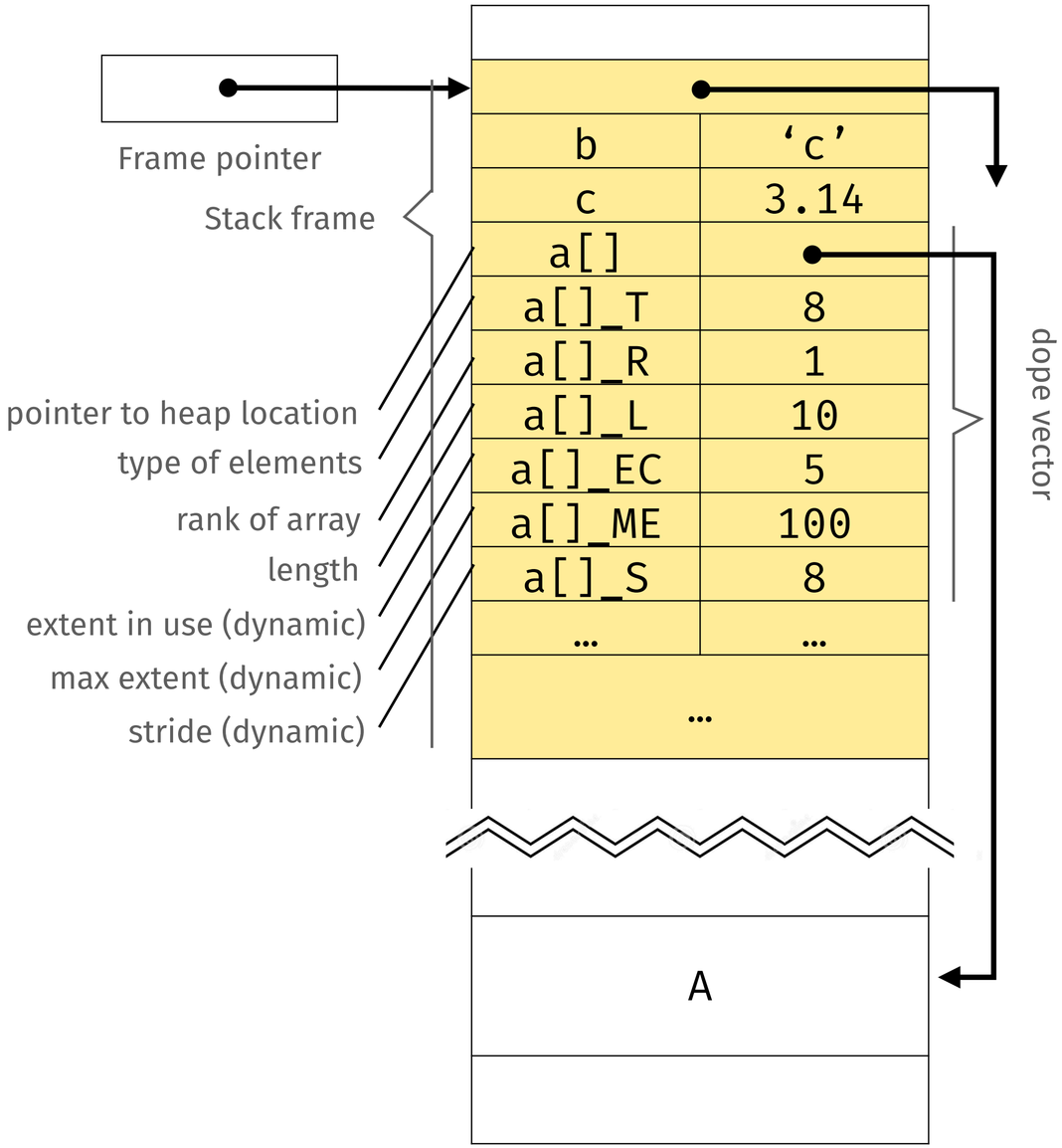
Se decidiamo di definire l'array in fase di compilazione, detto anche “in **forma statica**”, possiamo memorizzarlo nello stack frame del blocco che porta la sua definizione. In questo caso, conosciamo la dimensione necessaria per memorizzare l'array (l'offset tra il primo e l'ultimo elemento dell'array), quindi l'accesso a un elemento dell'array è simile all'accesso a variabili di tipo scalare (salvo alcuni calcoli necessari, ad esempio, quando si accede ad array multidimensionali).



Tipi Array

Se decidiamo di definire l'array **quando elaboriamo la sua dichiarazione**, conosceremo la sua forma (fissa) nel momento in cui il controllo raggiunge la dichiarazione dell'array. Un esempio di questo tipo è se la dimensione dipende dal valore di una variabile.

Possiamo allocare l'array nello stack frame del blocco che ne contiene la definizione. Tuttavia, poiché conosciamo la dimensione dell'array solo quando carichiamo il frame, non possiamo preallocare in modo sicuro lo spazio nello stack: una stima errata sprecherebbe memoria o si sovrapporrebbe ad altre variabili statiche. Per ovviare a questo problema, si utilizza **l'heap** e si memorizza nel frame il puntatore all'inizio della regione di memoria. Il descrittore di tale array prende il nome di **dope vector**, utilizzato anche nel caso di array **dinamici** (con alcuni elementi aggiuntivi nel dope vector per tenere traccia del suo stato).



Differenze tra i Tipi Array di C, Java, and Rust

Vediamo la sintassi di C, Java e Rust per dichiarare un array lineare di interi:

`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, la dichiarazione corrisponde all'allocazione (statica) dell'array, che possiamo utilizzare subito.

In Java, non creiamo l'array quando dichiariamo la sua variabile, ma (come ogni tipo Java non primitivo) il nome è un **riferimento**, e.g., a qualche valore di “array di interi”. Ad esempio, alla riga 3, assegniamo a x un nuovo (**new**) array (nell'heap).

Anche in Rust, la dichiarazione introduce solo l'annotazione del nome x, che in seguito viene associato a un array (statico) (riga 3). In questo caso, il tipo viene riportato nell'assegnazione per verificare il vincolo di dimensione.

Tipi Insieme

Un tipo di insieme denota una struttura di dati piatta e **senza ordine** con **valori unici dello stesso tipo**.

Le operazioni possibili sugli insiemi includono il **test di inclusione** e le comuni operazioni di manipolazione degli insiemi: **unione**, **intersezione**, **differenza** e **complemento**.

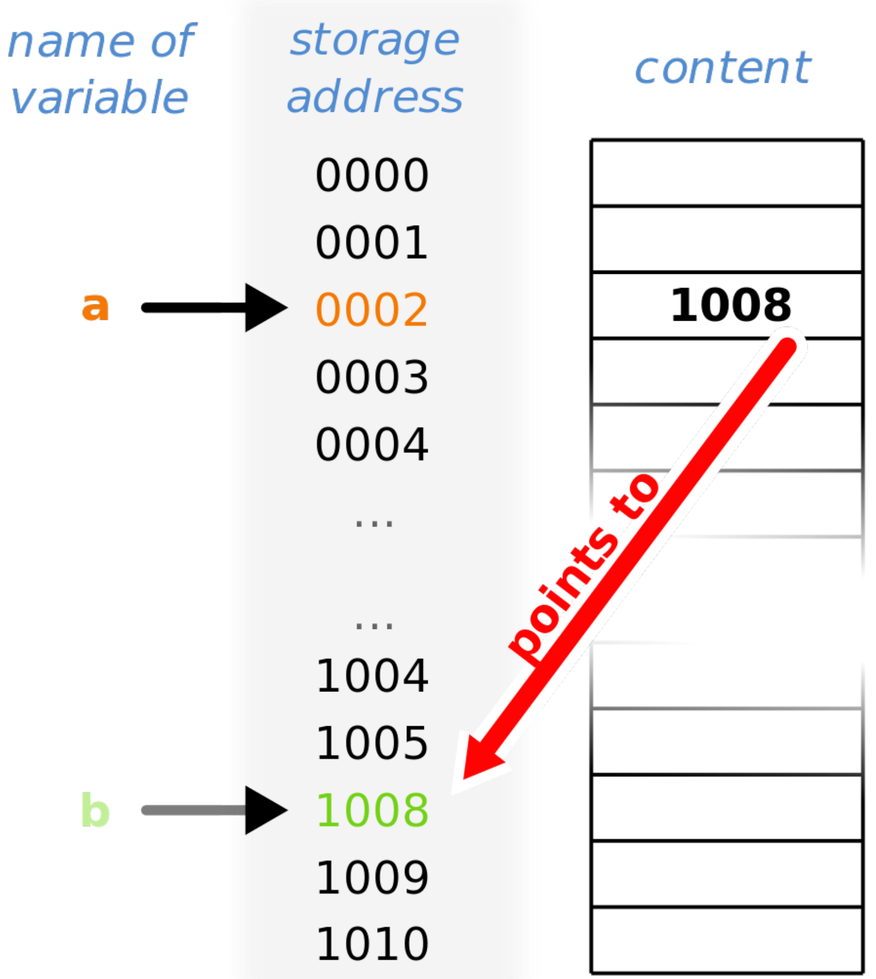
Un modo efficiente per rappresentare un insieme è un array di bit di lunghezza pari alla cardinalità del tipo di base. Questo array è chiamato **array caratteristico** e, in esso, il bit j-esimo indica se l'elemento j-esimo del tipo base (dato un ordinamento standard) appartiene all'insieme. Questa rappresentazione permette di eseguire in modo efficiente le operazioni sugli insiemi (operazioni bit-a-bit sulla macchina fisica), ma non è adatta a grandi sottoinsiemi di tipi base. Per ovviare a questo problema, i linguaggi spesso limitano i tipi che si possono usare come tipi base di un insieme oppure scelgono rappresentazioni alternative, ad esempio tramite tabelle hash, bilanciando velocità e occupazione di memoria.

Tipi Riferimento

Un riferimento dà accesso indiretto a un altro valore (ad esempio, eventualmente assegnato a una variabile), cioè **fa riferimento** a un dato.

Le operazioni tipiche supportate dai riferimenti sono la **creazione**, il **controllo di uguaglianza** e la **dereferenziazione**, cioè l'accesso al dato referenziato.

I riferimenti sono particolarmente presenti nei linguaggi di basso livello, dove vengono utilizzati per passare/condividere dati di grandi dimensioni o mutabili.

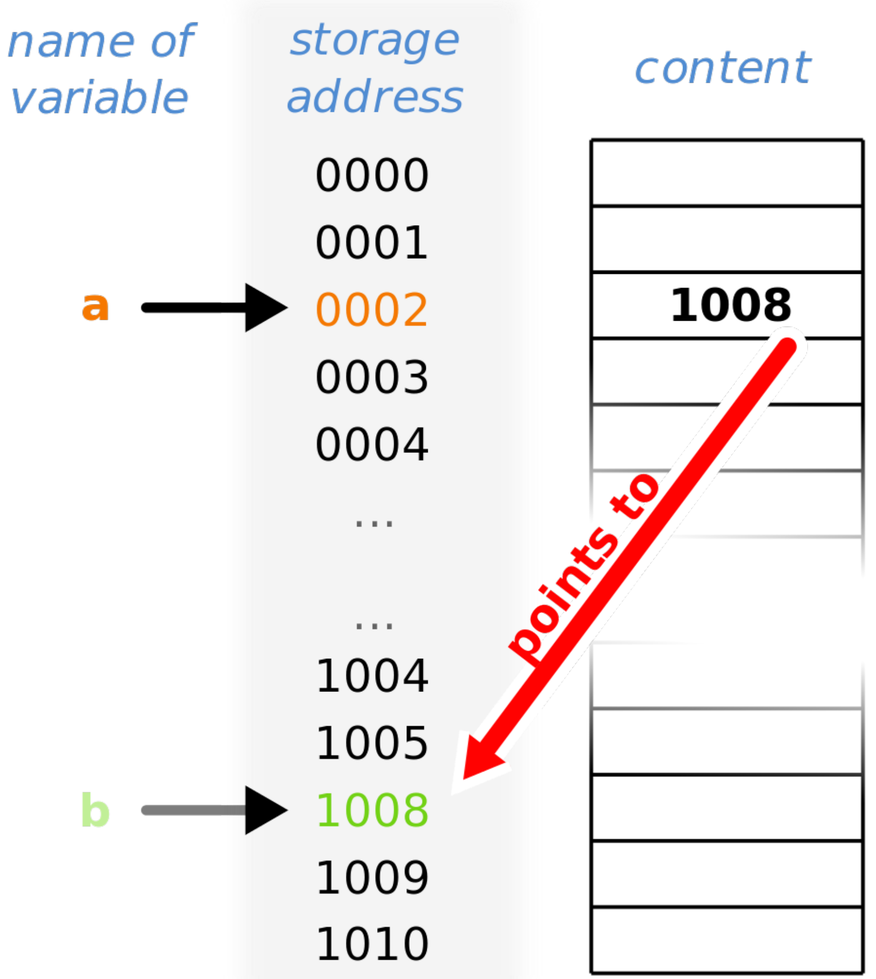


Tipi Riferimento

L'implementazione più comune dei riferimenti è quella dell'indirizzo fisico di memoria, il **puntatore**, del dato in memoria.

Tuttavia, i puntatori sono solo un'istanza dei riferimenti, che possono essere, ad esempio, **indici di array**.

I riferimenti possono fare riferimento a riferimenti, come nelle strutture di dati come gli alberi e le liste.

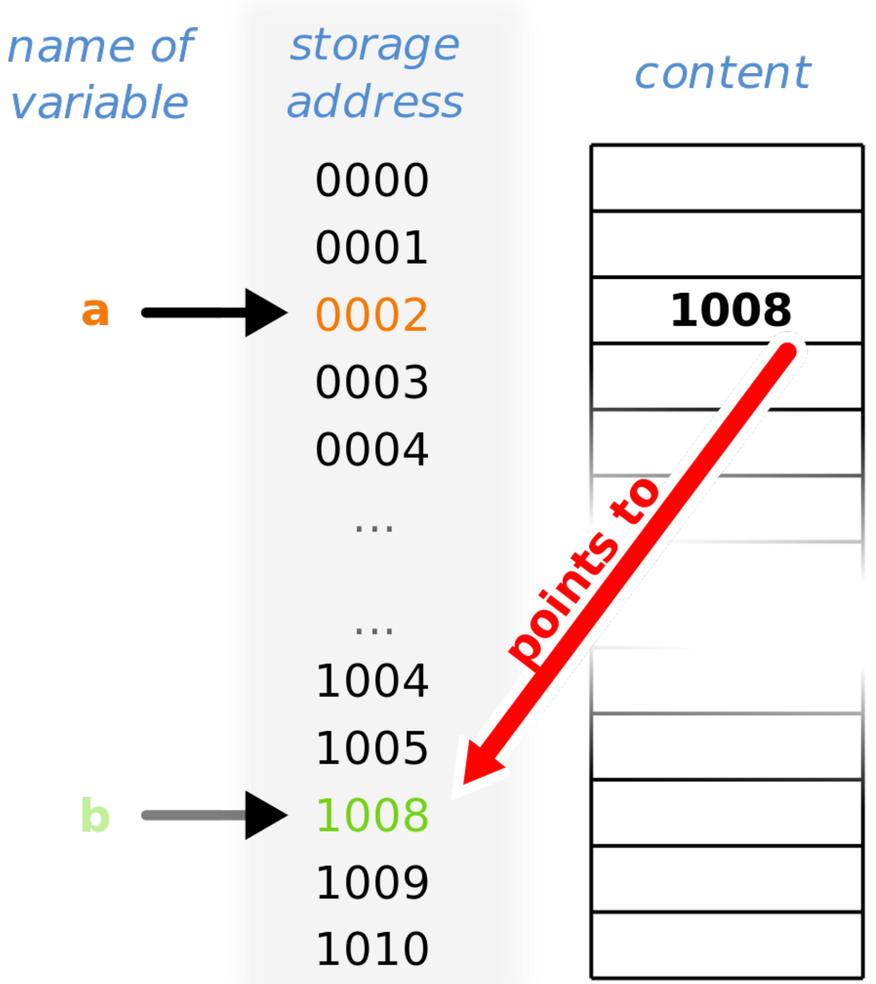


Tipi Riferimento

I riferimenti possono introdurre complessità nei programmi, poiché **chiedono al programmatore di pensare in termini dinamici** anziché statici.

Senza la dovuta diligenza o controlli dedicati, i riferimenti possono diventare:

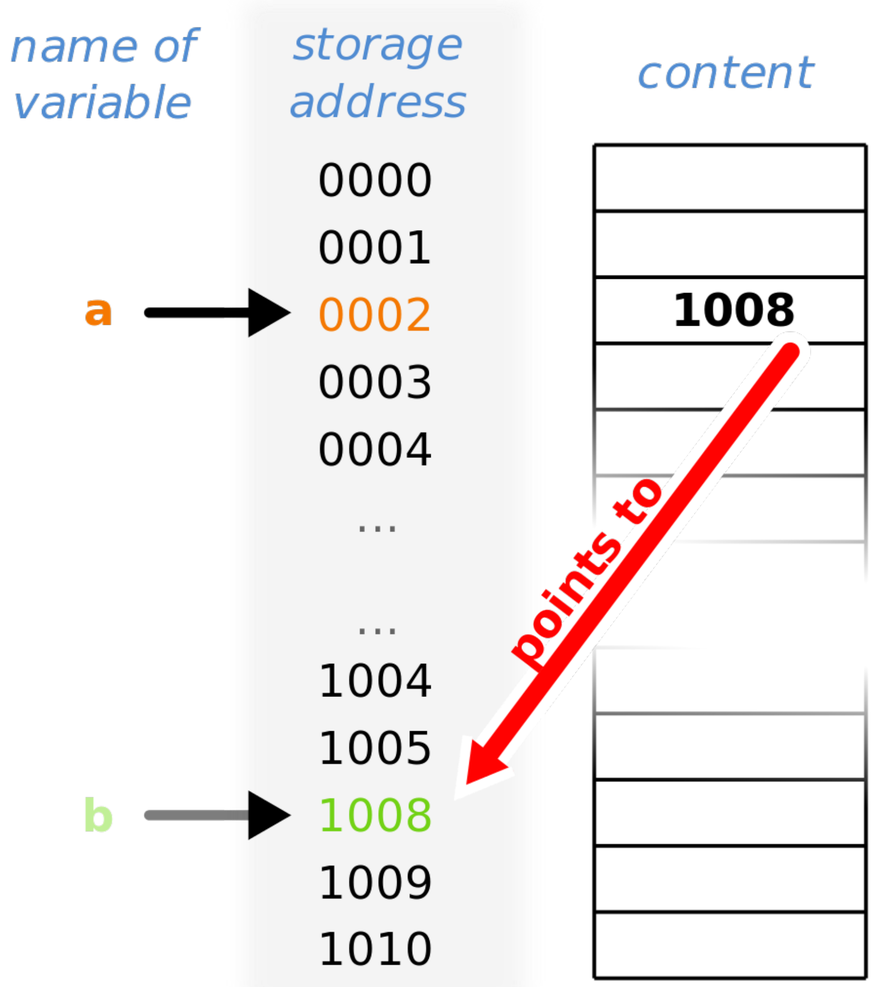
- "**wild**" quando, non inizializzati, il loro accesso può causare un comportamento inaspettato;
- "**dangling**" quando il dato referenziato è stato deallocato e l'accesso può portare a un comportamento inaspettato, soprattutto quando dati compatibili all'operazione in essere sovrascrivono il dato deallocato.



Tipi Riferimento

I linguaggi con un **modello di variabili di riferimento** raramente forniscono tipi di riferimento, poiché ogni variabile è sempre un riferimento (ad esempio, Java).

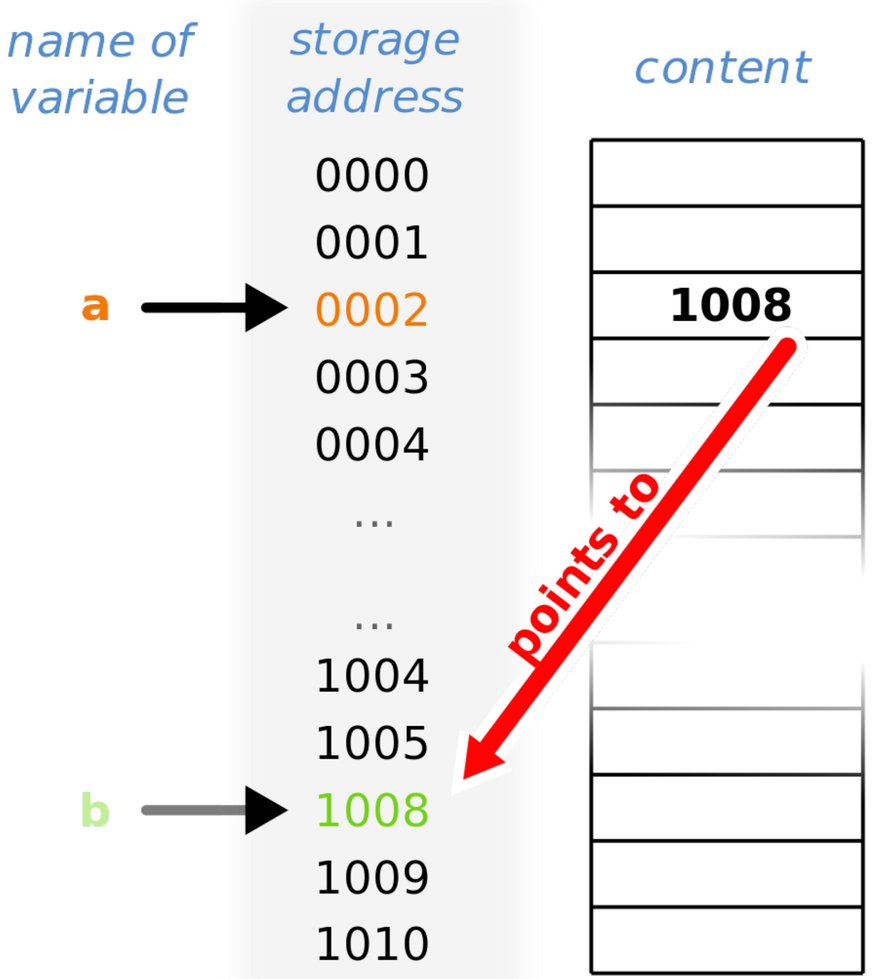
I linguaggi con variabili modificabili forniscono riferimenti che consentono al programmatore di fare riferimento ai valori senza dereferenziarli.



Tipi Riferimento

In C, `int* x` specifica un riferimento (puntatore) a una locazione di memoria (cioè a variabili modificabili) che contiene un valore di tipo intero.

A seconda del linguaggio (modello), i puntatori possono fare riferimento a posizioni arbitrarie o seguire alcuni vincoli. Ad esempio, Pascal richiede che i puntatori facciano riferimento a valori allocati sull'heap, mentre il C ammette puntatori che fanno riferimento allo stack o all'area globale.



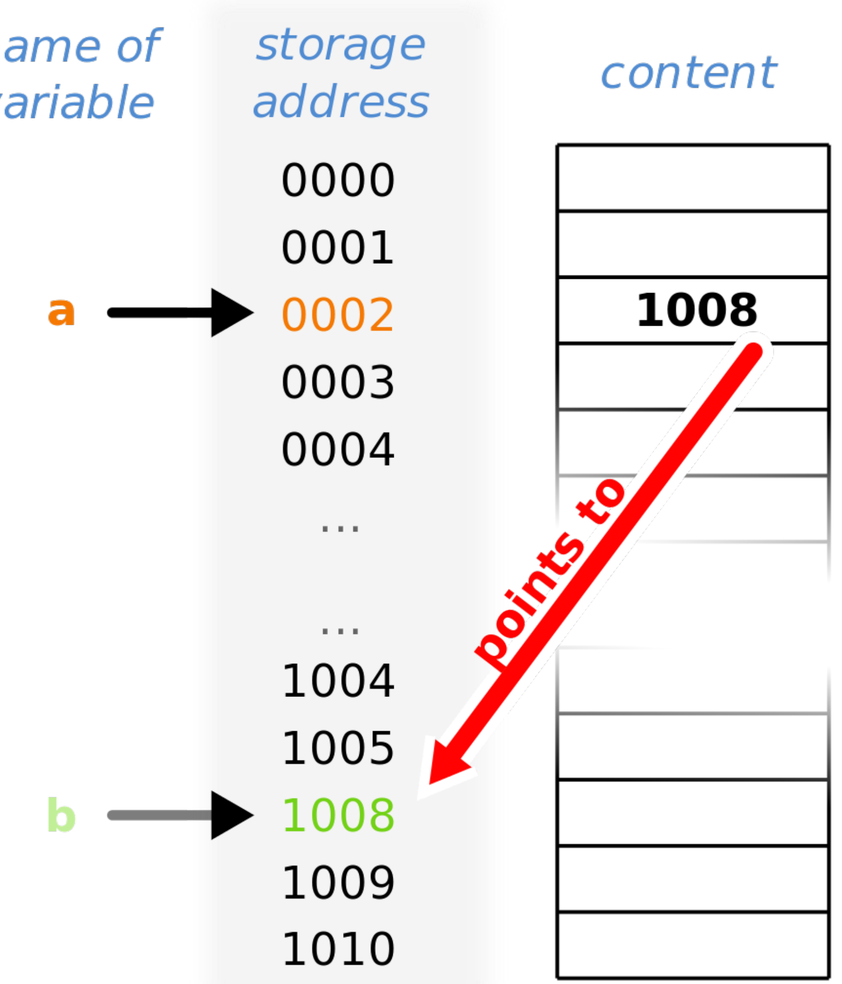
Tipi Riferimento

I linguaggi con riferimenti spesso definiscono un puntatore "**canonico**" che abita il tipo riferimento (lo stesso, per qualsiasi tipo associato): **`null`**. Il punto di avere null nel linguaggio è utilizzarlo per “invalidare” un puntatore a runtime. Ad esempio il seguente codice C è un tipico esempio di invalidazione e inizializzazione esplicita dei puntatore.

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

Sopra, p viene creato ma non è inizializzato implicitamente e non è sicuro che sia valido. Per questo motivo, lo inizializziamo esplicitamente a NULL, fintanto che non allochiamo effettivamente la memoria nel programma.

Qui usiamo `malloc` per allocare una quantità specifica di byte (la `sizeof(int)`, sopra) nello heap. Poiché `malloc` ignora i tipi, restituisce sempre un puntatore nullo (`*void`), a indicare che si riferisce a una regione di memoria di tipo sconosciuto.



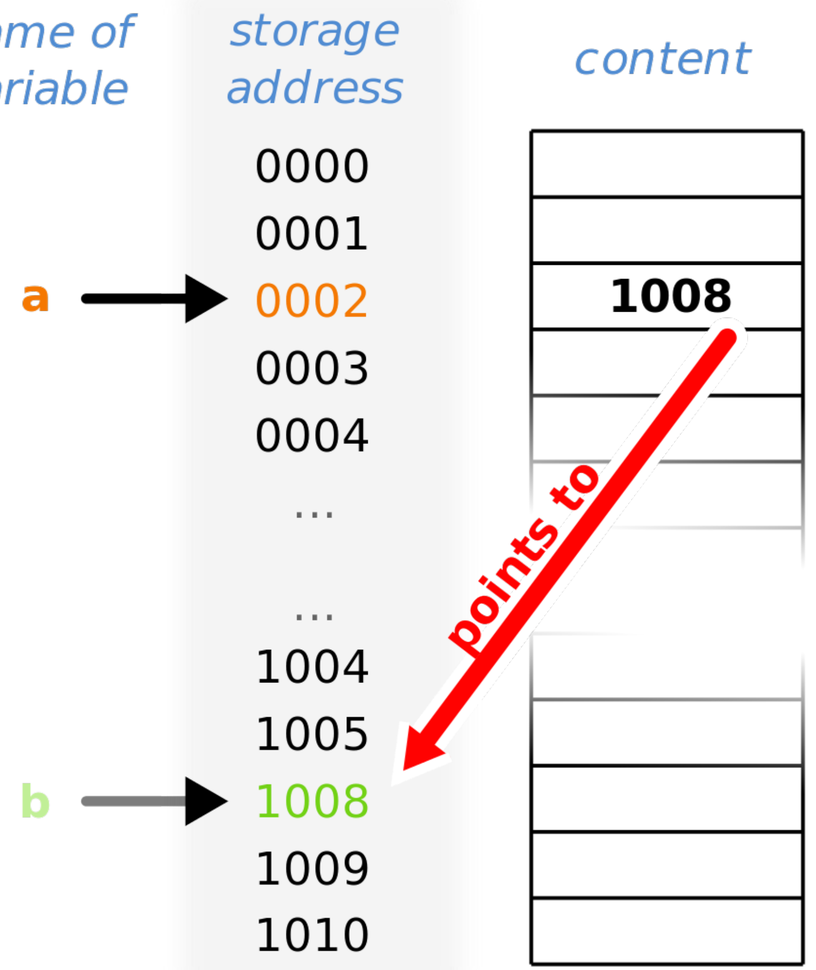
Tipi Riferimento

I linguaggi con riferimenti forniscono un **operatore di riferimento alle variabili**, cioè per creare un riferimento alla posizione in memoria della variabile. Ad esempio, `&` in C

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

Il puntatore `p` punta alla posizione che contiene la variabile `pi`.

A differenza di `malloc`, la referenziazione sopra consente al puntatore di fare riferimento a posizioni nello stack.



Tipi Riferimento

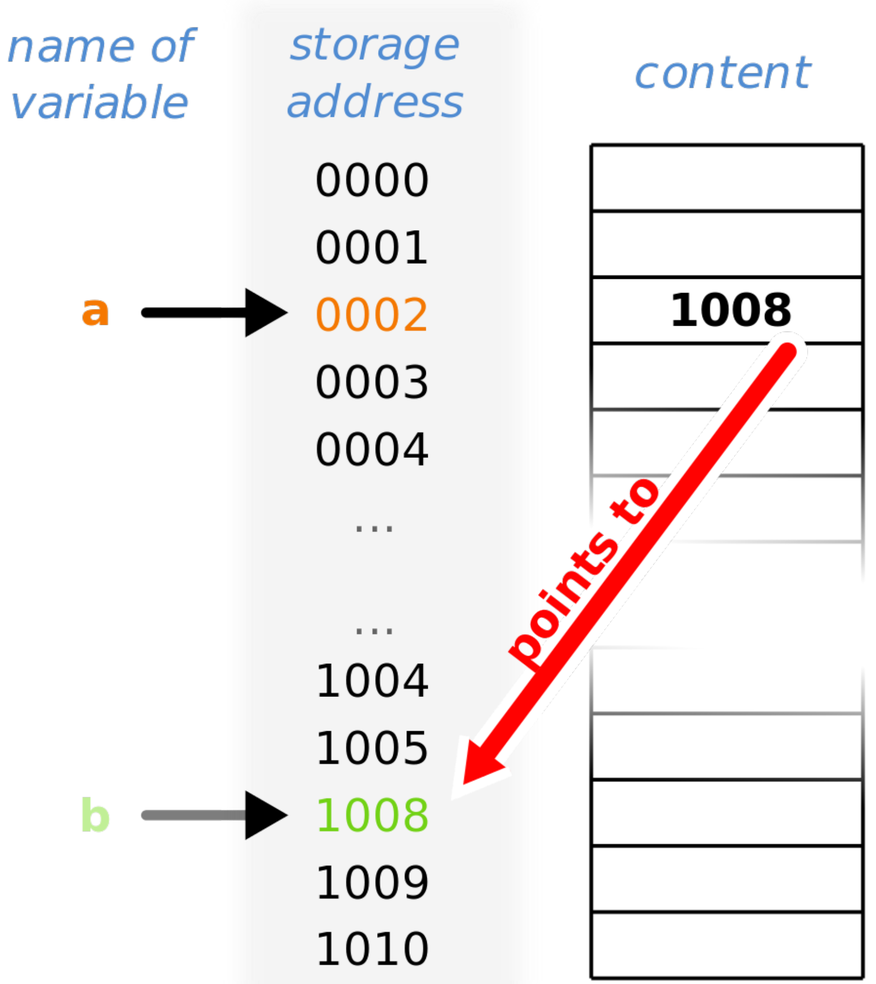
I linguaggi con riferimenti prevedono anche un operatore di **dereferenziazione**, ad esempio * in C

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

dove assegniamo il valore `4,1415` a `pi`, dereferenziando p sia a sinistra che a destra dell'assegnazione.

Nell'ultima riga, la dereferenziazione a destra di = ci permette di *leggere* il contenuto della locazione referenziata da p (il contenuto della variabile `pi`), mentre la dereferenziazione a sinistra di = ci permette di *scrivere* sulla locazione referenziata da p (quella corrispondente a `pi`).

Si noti che l'assegnazione non modifica il valore di p, poiché viene sempre utilizzato nella sua forma dereferenziata.



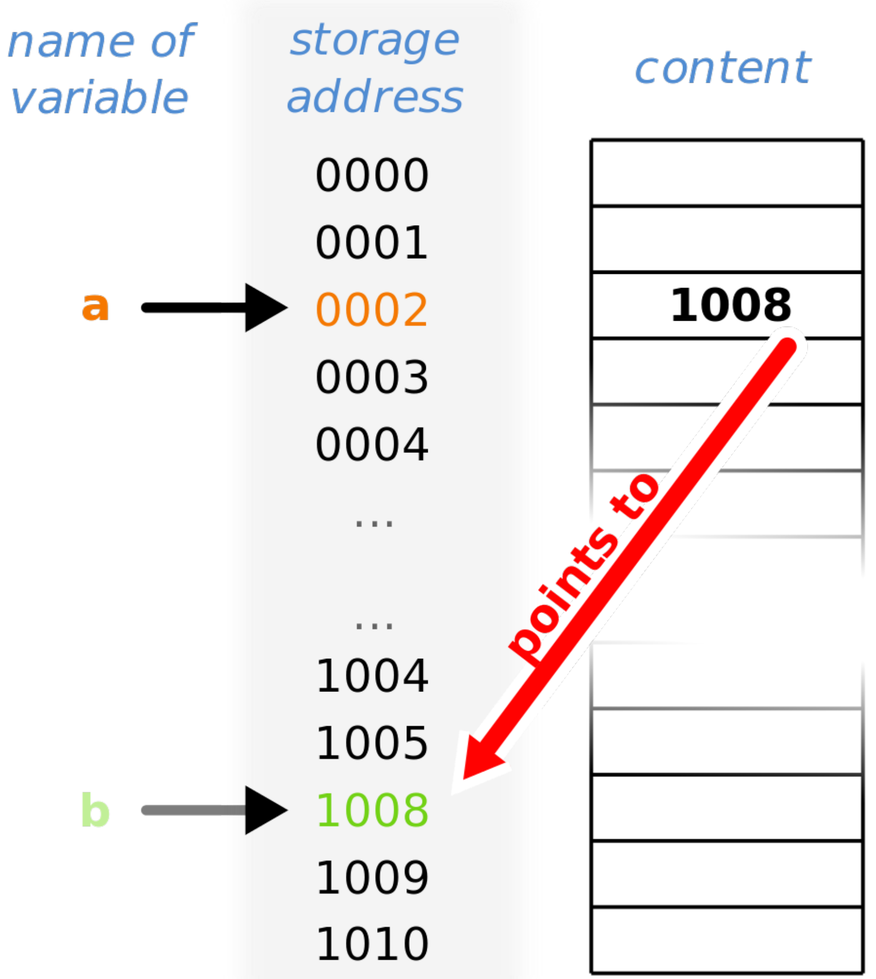
Tipi Riferimento

Un linguaggio con riferimenti può incorrere in una **deallocazione implicita**, ad esempio (in C)

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

possiamo creare una regione di memoria non accessibile, poiché abbiamo distrutto l'unico puntatore per raggiungerla.

Questa porzione di memoria “non liberata” può crescere nel tempo (finché il programma viene eseguito) e può incorrere in un fenomeno chiamato “memory leak”. Il problema del recupero di queste porzioni di memoria è stato oggetto di studi in diverse direzioni, dai *garbage collector* (ad esempio, Java) ai sistemi di tipi che impediscono questo tipo di comportamenti (ad esempio, Rust).

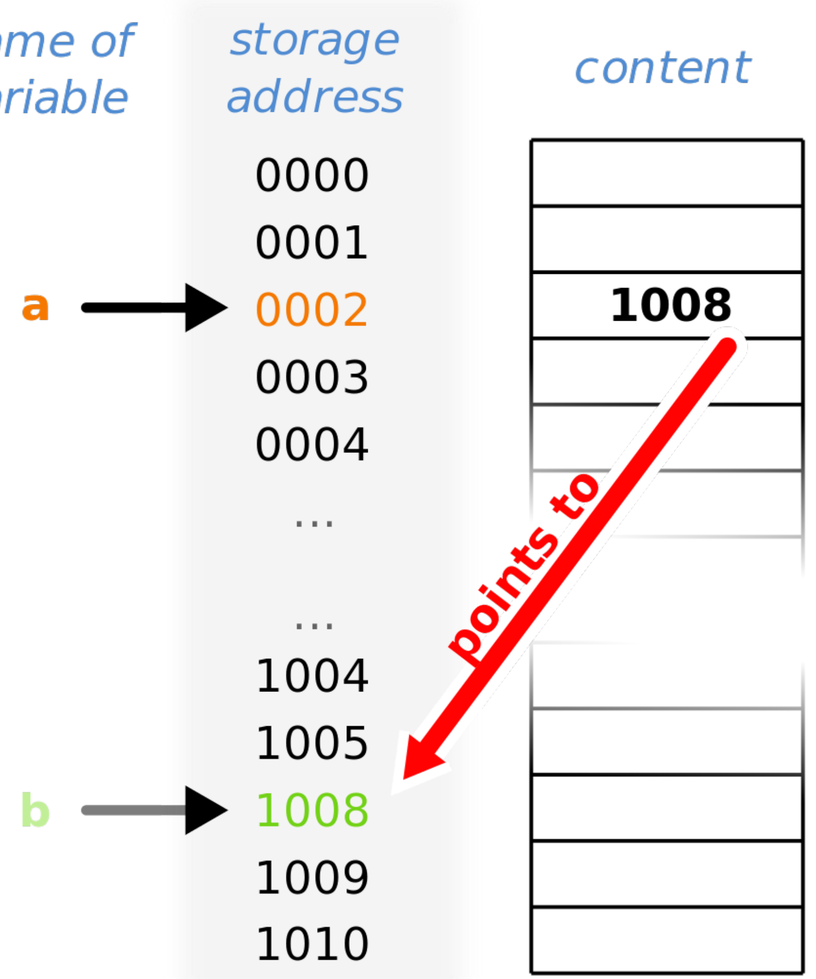


Tipi Riferimento

I linguaggi con riferimenti forniscono un **operatore di deallocazione esplicito** per rilasciare la memoria referenziata da un puntatore. C fornisce `free`

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

Come nel caso dei puntatori non inizializzati, è buona norma NULLificare un puntatore liberato. Chiamare `free` su un puntatore allo stack è un errore semantico (potrebbe portare a un comportamento imprevedibile).



Tipi Riferimento, Rust

Rust è noto per la sua **gestione sicura dei puntatori**. Il linguaggio fornisce gli stessi operatori di C, ma mette in atto controlli statici che consentono al compilatore di liberare automaticamente la memoria inutilizzata e di prevenire riferimenti nulli, puntatori dangling, double-free e invalidazioni di puntatori.

```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
>
```

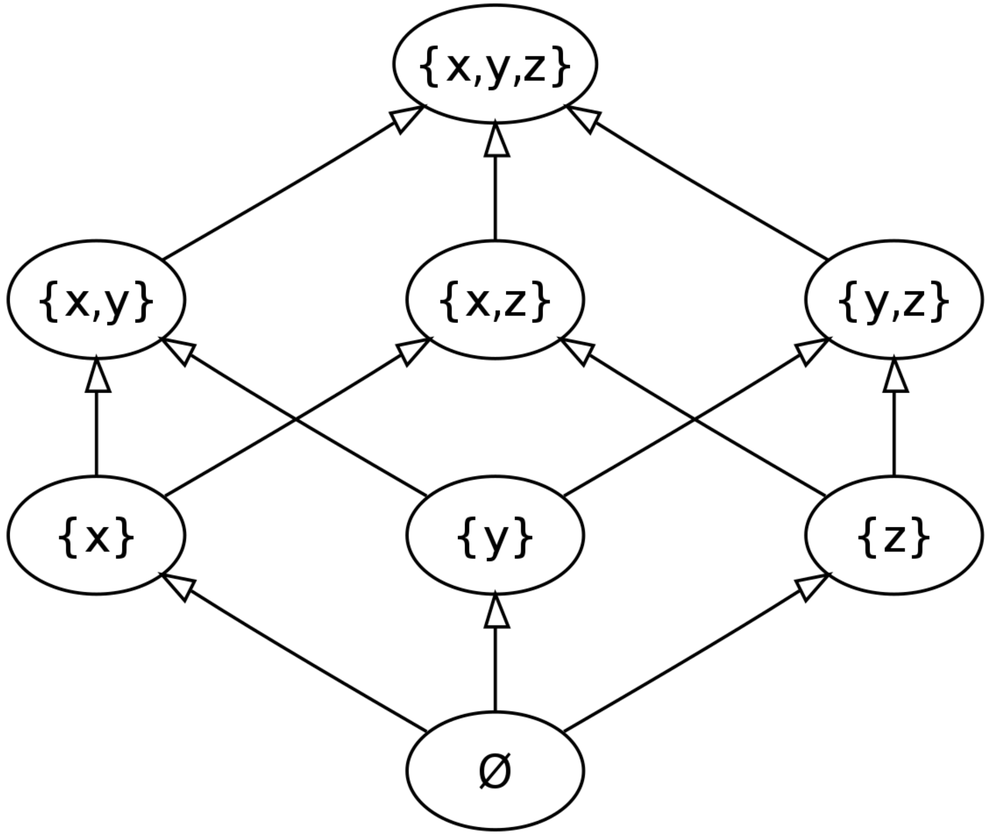
`&T`	Consente a uno o più riferimenti di leggere T
`&mut T`	Consente a un singolo riferimento di leggere e scrivere T
`*const T`	Accesso in lettura non sicuro a T
`*mut T`	Accesso non sicuro in lettura e scrittura a T
`Box<T>`	T allocato su heap con un unico proprietario che può leggere e scrivere T
`Rc<T>`	T allocato su heap con molti lettori
`Arc<T>`	Come una Rc<T>, ma con garanzie di sicurezza per il multi-threading

Insiemi potenza (verso i tipi prodotto)

Assumiamo l'assioma dell'insieme di potenza, cioè

$$\wp(S) = \{T : T \subseteq S\}$$

Chiamiamo l'insieme $\wp(S)$ **l'insieme di potenza** (o power-set) di S, che contiene tutti i sottoinsiemi generati da qualsiasi combinazione (anche vuota) di elementi in S.



Paia Ordinate e Prodotto Cartesiano (verso i tipi prodotto)

Sia $a \in A$ e $b \in B, A \cap B = \varnothing$ e sia $(a,b) \triangleq \big\{\{a\},\{a,b\}\big\}$ allora

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

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

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

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

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

Sia $\{(a,b) \,|\, a \in A \wedge b \in B\} \triangleq A \times B$

chiamato **prodotto cartesiano** di A e B

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

Tipi Prodotto

Array, set e puntatori sono esempi di tipi compositi che "prendono come parametro" un tipo.

Quando si combinano due o più tipi in una struttura fissa, si parla di **tipi prodotto**.

Il nome deriva dalla nozione di "prodotto diretto" della matematica, che è una generalizzazione del prodotto cartesiano $A \times B \triangleq \{(a, b) \mid a \in A \wedge b \in B\}$.

I tipi di prodotto più comuni sono **coppie** (**pair**), **tuple**, **record** e **varianti**.

Per convenzione, il prodotto vuoto è l'**unità**.

Tipi Prodotto • Coppie (Pair) e Tuple

La forma più semplice di tipo prodotto è la **coppia**. Dati due tipi A e B, il tipo $A \times B$ denota l'insieme delle coppie (le possibili combinazioni) dei valori in A e B. La generalizzazione delle coppie sono le **tuple**, che definiscono il prodotto di un numero arbitrario e finito di tipi $T_1, \cdots, T_n$ come $\prod_{i=1}^{n} T_i = T_1 \times \cdots \times T_n$

C e Java non supportano direttamente coppie/tuple (vedremo che si possono implementare usando i *record*). Rust supporta le tuple con la sintassi `(T1, ... ,Tn)`, ad esempio `(i32,i32)` è una coppia di interi che può rappresentare un sistema di coordinate. Poiché le tuple definiscono tipi basati sull'ordine dei suoi componenti, Rust segue questa astrazione per accedere ai componenti di un valore tupla, ad esempio, `coords.0`

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

Tipi Prodotto • Record

I record interpretano i tipi prodotto sostituendo la definizione posizionale dei componenti con la loro identificazione mediante etichette (distinte). I linguaggi implementano i record in modi diversi, ad esempio come **strutture** (struct alla C/ Rust) o **classi** (Java 17 ha introdotto i **record**). Gli elementi di un record sono chiamati **campi** (**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;
```

Tipi Prodotto • Record

Nei record, l'ordine dei campi può essere significativo e venire seguito nella loro rappresentazione in memoria, ad esempio, memorizzando i campi in posizioni contigue, anche se l'allineamento bit-a-bit può comportare degli spazi vuoti tra i campi. Ad esempio, per Rust e C il `name` di `Person` richiederà sempre 5 byte. Mentre in C questo non è evidente, Rust ci costringe a "riempire" sempre i possibili valori mancanti con valori predefiniti (ad esempio, i valori vuoti di char per “Eva”). Poiché Java alloca gli oggetti nell'heap, il problema non si presenta (e non specifichiamo, nel tipo, vincoli sulla dimensione di `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;
```

Tipi Prodotto • Pattern Matching

Se da una parte i tipi prodotto producono nuovi tipi di dati strutturati, abbiamo bisogno di costrutti che rendono semplice destrutturarli. Uno di questi costrutti è il **pattern matching**.

Il pattern matching controlla e individua elementi specifici rispetto a un certo schema, ad esempio in Rust

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

Le implementazioni safe di pattern matching garantiscono una corrispondenza esaustiva, che aiuta a escludere errori comuni come i casi mancanti, i casi impossibili e i casi ridondanti.

Tipi Prodotto • Pattern Matching

Il pattern matching controlla e individua elementi specifici rispetto a un certo schema, ad esempio 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; // implicit pattern-matching
let evaR = PersonR{ name, age: match age {
    1..=10 => [ 'K','i','d','' ],
    11..=20 => [ 'T','e','e','n' ],
    _ => [ 'O', 'l', 'd', '' ]
}}
```

Tipi Prodotto • Tipi Ricorsivi

I tipi ricorsivi (come concetto) sono utili per definire strutture di dati, come liste e alberi, che possiamo modificare dinamicamente. I tipi prodotto sono un modo per esprimere tipi **ricorsivi**. Nell'esempio seguente, in Java, si noti l'uso di `null` per “chiudere” la struttura.

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

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

Tipi Somma

I tipi prodotto descrivono composizioni di tipi. Ci possono essere casi in cui si vuole indicare che **una variabile può contenere un'unione disgiunta di tipi**, ad es,

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

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

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

Quindi, dichiarare che x è di tipo int $\sqcup$ char significa che x può contenere un intero o un carattere. L'unione degli **insiemi etichettati** ($\text{int}^* \cup \text{char}^*$) ci dice che, anche se ci possono essere elementi coincidenti negli insiemi, sappiamo sempre a quale insieme appartengono originariamente quei valori (ad esempio, $(0, c)$ e $(0, i)$).

Tipi Somma

Le unioni disgiunte di tipi sono solitamente chiamate **tipi somma** (ma anche **unioni etichettate/tagged union**, **tipi unione**, **tipi scelta**, **tipi variante** e **coprodotti**).

Un esempio pratico di tipo somma è quello di un tipo `Address`, in grado di spaziare su entrambi i tipi `PhysicalAddress` e `VirtualAddress` (ad esempio, l'indirizzo postale di una persona e la sua e-mail).

Alcuni linguaggi (in particolare quelli ispirati a Pascal e ML) forniscono un supporto diretto ai tipi di somma tramite un operatore dedicato, ad es,

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

Tipi Somma

Oltre ad avere operatori espliciti nel linguaggio, abbiamo visto un tipo che può aiutare a definire insiemi di valori: le enumerazioni. Linguaggi come Java e Rust hanno esteso le enumerazioni per catturare il caso dei tipi di somma. Ad esempio,

```
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 } => …
}
```

Tipi Somma

Oltre a utilizzare le enumerazioni (come in Rust), Java ha recentemente introdotto le **sealed classes**, che definiscono le uniche strutture di dati che possono apparire come uno dei possibili tipi presenti in un dato tipo di somma.

```
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 -> …
}
```

Tipi Somma

Il C ha una nozione di unione di strutture. Le **union** in C sono un modo per far sì che la stessa posizione di memoria contenga diversi tipi di dato (ad esempio, un intero o un carattere), dove la memoria viene allocata in base alla struttura più grande (nell'esempio a fianco, la dimensione del carattere).

Tuttavia, il linguaggio non disciplina il modo in cui gli utenti interagiscono con (la posizione di una) variabile union. Ad esempio, data una variabile x di tipo union, possiamo scriverci un intero e poi leggerlo come char, senza che il compilatore sollevi alcun errore/avvertimento.

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

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

// data.i = 65
```

Tipi Somma • Tipi Ricorsivi

I tipi somma sono un'alternativa ai tipi prodotto per i tipi ricorsivi (omettendo **null**!)

```
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() ) ) );
```

Relazioni (verso le Funzioni)

Data una sequenza di insiemi $S_1, \ldots, S_n$, chiamiamo l'insieme $\mathbb{R} \subseteq S_1 \times \cdots \times S_n$ una **relazione** sul prodotto cartesiano $S_1 \times \cdots \times S_n$ quando $\mathbb{R}$ relaziona gli elementi $s_1 \in S_1, \cdots, s_n \in S_n$, cioè, quando, per qualche $s_1, \cdots, s_n$, $(s_1, \cdots, s_n) \in \mathbb{R}$.

Quando $\mathbb{R} \subseteq S \times T$ diciamo che $\mathbb{R}$ è una **relazione binaria** (spesso usato anche per indicare il caso speciale $\mathbb{R} \subseteq S \times S$). Dato $s \in S$ e $t \in T$ se $(s, t) \in \mathbb{R}$ scriviamo solitamente $s\ \mathbb{R}\ t$. Per convenzione, dato $\mathbb{R} \subseteq S \times T$, chiamiamo gli elementi di S in $\mathbb{R}$ il **dominio** di $\mathbb{R}$ ($\text{dom}(\mathbb{R}) \subseteq S$) e gli elementi di T in $\mathbb{R}$ **immagine** di $\mathbb{R}$ ($\text{img}(\mathbb{R}) \subseteq T$)

Funzioni

Quando $\mathbb{R} \subseteq S \times T$, $\forall t, t'\ s\ \mathbb{R}\ t, s\ \mathbb{R}\ t'$ e $t = t'$ allora chiamiamo $\mathbb{R}$ una **funzione parziale** — "funzione" perché mappa un valore del dominio a un unico del co-dominio, "parziale" perché non assumiamo che $\mathbb{R}$ consideri la totalità dei valori in S.

Quando $\mathbb{R}$ è una funzione parziale, solitamente adottiamo la notazione $\mathbb{R}(s) = t$ (alternativa a $(s, t) \in \mathbb{R}$ e $s\ \mathbb{R}\ t$) e chiamiamo s l'**argomento** di $\mathbb{R}$ e t il **valore** di $\mathbb{R}$ per s. Diciamo anche che $\mathbb{R}$ **mappa** s a t e adottiamo una notazione alternativa $\mathbb{R} \subseteq S \times T \equiv \mathbb{R} : S \to T$

Quando $\text{dom}(\mathbb{R}) = S$ diciamo che $\mathbb{R}$ è una **funzione totale**.

Funzioni

Dato $f \subseteq S \times T$, tale che $f \in \wp(S \times T)$, visto che $S \times T \subset \wp(\ \wp(S \cup T)\ )$ e $S \times T \in \wp(\ \wp(\ \wp(S \cup T)\ )\ )$, sia $\wp(S \times T) \triangleq T^S$, allora $f \in T^S$, cioè f appartiene ad una famiglia di relazioni/funzioni che legano elementi di S a elementi di T. La cardinalità di tale famiglia è $|T|^{|S|}$

Ad esempio, se $S = \{1,2,3\}$ e $T = \{f, t\}$, il numero di possibili $g : S \to T$ è 2^3 $(\ (f,f,f), (t,f,f), (t,t,f), (f,t,f),\ \ldots, (t,t,t)\ )$.

Data la definizione sopra, abbiamo un modo alternativo di scrivere $\wp(S)$ come 2^s. Questo è possibile se consideriamo $2 \triangleq \{0,1\}$ e $\wp(S)$ come descritto dalla famiglia di funzioni che associano set di elementi di S alla loro presenza in $\wp(S)$, e.g., sia $S = \{1,2,3\}$ allora $\wp(S)$ è descritto dalle funzioni $f_0 = \{(1,0),(2,0),(3,0)\} = \varnothing$, $f_1 = \{(1,1),(2,0),(3,0)\} = \{1\}$, $\ldots$, $f_8 = \{(1,1),(2,1),(3,1)\} = \{1,2,3\}$

Tipi Funzione

I linguaggi di alto livello supportano spesso la definizione di funzioni (o procedure), tuttavia solo alcuni denotano il tipo di funzioni (cioè danno loro un nome nel linguaggio). Ad esempio, se `f` è una funzione definita come `R f(P p){ ... }`, possiamo indicare il suo tipo come $P \to R$, dove `P` è il tipo del parametro unario accettato da `f` e `R` è il tipo del valore restituito da `f`. La rappresentazione dalla teoria insiemistica di $P \to R$ è R^P.

Questa disciplina di denominazione segue la poliadicità delle funzioni, ad esempio una funzione di forma

`R f( P1 p1, ... , Pn pn ){ ... }` ha tipo $P1 \to \ldots \to Pn \to R$ o $R^{P1 \cdots Pn}$

I valori di un tipo di funzione sono denotabili in tutti i linguaggi, ma solo alcuni (i cosiddetti linguaggi “funzionali”) li rendono esprimibili (o memorizzabili). La principale operazione consentita su un valore di tipo funzione è **l'applicazione**, cioè l'invocazione di una funzione su alcuni argomenti (parametri reali).

L'Algebra dei Tipi

I tipi Prodotto, Somma e Funzione richiamano l'esistenza di un'algebra — una disciplina che definisce le regole per la manipolazione dei simboli (di tipo) dei tipi, definiti quindi "**tipi** (di dati) **algebrici**".

I sistemi di tipi possono utilizzare le proprietà di questa algebra per esprimere e verificare le proprietà dei programmi.

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 (number of) functions from the combinations of B given A, e.g., Bool -> Unit (1^2) and Unit -> Bool (2^1)

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*.

Equivalenza di tipo

Una delle domande principali che possiamo porci sui tipi di un programma è

“quando sono uguali due tipi?”

che è alla base di alcuni dei test di correttezza del type checking.

Le risposte alle domande sull'uguaglianza non hanno un'unica interpretazione, poiché possono dipendere dal contesto in cui si verifica l'uguaglianza.

Per esempio, sia S un tipo definito come un **sottoinsieme** di numeri interi e `f` una funzione che può sommare due numeri interi qualsiasi. Dal suo punto di vista, **`f` può considerare gli abitanti di S** (e quindi il tipo, S, per estensione) **come equivalenti agli interi**, poiché “sa” che può lavorare su un sottoinsieme dei valori che accetta. Al contrario, se `f` accettasse solo valori di S, non potremmo assumere con sicurezza i parametri interi come uguali a S, poiché, ad esempio, il corpo della funzione potrebbe considerare qualche **invariante** di S, invalidata dagli interi (e.g., gli abitanti di S sono sempre positivi).

Preordini e equivalenze

Dato un insieme S, chiamiamo l'insieme $\mathbb{R}$ una relazione binaria su S quando $\mathbb{R}$ è un sottinsieme del prodotto cartesiano di S per sé stesso, cioè $\mathbb{R} \subseteq S \times S$. Dati due elementi $\{s_1, s_2\} \subseteq S$ diciamo che questi sono nella relazione, scritto $s_1 \ \mathbb{R} \ s_2$, se $(s_1, s_2) \in \mathbb{R}$. Le relazioni binarie hanno diverse proprietà:

- **Riflessività**: dato s, $(s, s) \in \mathbb{R}$;
- **Simmetria**: dati s_1 e s_2, $\{(s_1, s_2), (s_2, s_1)\} \subseteq \mathbb{R}$;
- **Anti-simmetria**: dati s_1 e s_2 $\{(s_1, s_2), (s_2, s_1)\} \subseteq \mathbb{R}$ implica che $s_1 = s_2$;
- **Transitiva**: dati s_1, s_2, e s_3 $\{(s_1, s_2), (s_2, s_3)\} \subseteq \mathbb{R}$ implica $(s_1, s_3) \in \mathbb{R}$

Quando $\mathbb{R}$ è riflessiva e transitiva, chiamiamo $\mathbb{R}$ un **preordine**. Quando $\mathbb{R}$ è un preodine simmetrico, lo chiamiamo una **equivalenza**. Quando $\mathbb{R}$ è un preordine anti-simmetrico lo chiamiamo un **ordine parziale**.

Equivalenza di Tipo Nominale

Nei sistemi di tipi che considerano una nozione **nominale** di equivalenza dei tipi (NTE), ogni nuova definizione di tipo introduce un nuovo **nome** diverso da quelli esistenti.

Sia name$(T) = n$ una funzione che dato un tipo T ritorna il suo nome associato n, allora

$$T_1 \; NTE \; T_2 \iff \text{name}(T_1) = \text{name}(T_2).$$

Quindi, sebbene i tipi `Dollaro=int` ed `Euro=int` siano funzionalmente indistinguibili, in un sistema nominale non sono equivalenti.

Sebbene sia piuttosto semplice, l'idea alla base del sistema nominale è quella più aderente alle intenzioni del programmatore: ad esempio, se il programmatore usasse i tipi per distinguere tra Dollari ed Euro, potrebbero esserci alcuni invarianti (ad esempio, i tagli delle monete) non catturati a livello dei tipi su cui il programmatore fa affidamento nel corpo delle funzioni.

Duck Typing

Sebbene il controllo di tipo nominale sia solitamente eseguito in modo statico, sappiamo che i controlli di tipo possono avvenire anche a tempo di esecuzione (il caso principale è quello dei linguaggi interpretati).

Un modo alternativo e molto popolare di eseguire il controllo dei tipi a tempo di esecuzione è il cosiddetto metodo del **duck typing**, che funziona controllando se un dato valore supporta gli operatori previsti dal programma.

```
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'
```

Equivalenza di Tipo Strutturale

L'esempio del duck typing ha introdotto un'alternativa all'equivalenza di tipo nominale: finché non si osservano *differenze strutturali* tra i valori, possiamo considerarli dello stesso tipo.

In generale, questa interpretazione prende il nome di **equivalenza di tipo strutturale** (STE) e può essere eseguita anche staticamente. In questo caso, poiché non sappiamo in anticipo quali percorsi prenderanno i valori nel programma, dobbiamo eseguire **controlli più conservativi** di quelli "operativi" visti per il duck typing: verifichiamo l'equivalenza dei tipi confrontando tutte le loro operazioni, strutture e sottoelementi.

Naturalmente, questo rende la definizione di equivalenza strutturale più complessa, ad es,

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

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

Adozione di Sistemi di Tipo Nominale vs. Strutturale

Oltre a incarnare una nozione rigorosa di equivalenza, il tipaggio nominale presenta diversi altri vantaggi:

- un collegamento diretto per il runtime, ad esempio, alla **stampa**, al **marshalling** e alla verifica della **coercizione** dei tipi (spoiler);
- una **denotazione intuitiva dei tipi ricorsivi**—tipi la cui definizione fa riferimento al tipo stesso, come le liste e gli alberi (per esempio, una List di List o anche tipi mutuamente ricorsivi, per esempio, Tree di List);

$$\text{IntList} := (\text{int} \times \text{IntList}) + \text{Unit} \quad \text{ListIntList} := (\text{ListInt} \times \text{ListIntList}) + \text{Unit} \; vs \; \mu t \,.\, ((\mu t' \,.\, (\text{int} \times t') + \text{Unit}) \times t) + \text{Unit}$$

- il controllo della **sottotipaggio** (spoiler) è un **controllo diretto**, quasi banale, delle relazioni nominali di sottotipo dichiarate tra i tipi nominati.

Questi vantaggi hanno decretato il “successo” dei sistemi di tipi nominali, presenti in molti linguaggi di programmazione mainstream, ad esempio Java e Rust. Anche il C ha un sistema di tipi nominali di rilievo, sebbene la dichiarazione di `typedef` consenta agli utenti di equiparare tipi diversi con alias coincidenti (questa caratteristica è solitamente considerata non sicura, ad esempio, ricordiamo l'esempio di Dollari e Euro).

Compatibilità di tipo

L'esempio del duck typing ha mostrato che possiamo usare correttamente una struttura contenente i campi x, y e z in una funzione sum che richiede solo valori con i campi x e y. Questo significa che a volte possiamo usare una versione indebolita dell'equivalenza e ottenere comunque un programma corretto.

Questa forma indebolita di equivalenza viene solitamente chiamata **compatibilità di tipo**.

Formalmente, poiché l'equivalenza è un *preordine simmetrico*, questa sussume la compatibilità, che è un *preordine* (riflessivo e transitivo), ma non viceversa (non tutti i tipi compatibili sono equivalenti).

Compatibilità di tipo

La relazione di compatibilità varia a seconda dei linguaggi. Alcune interpretazioni comuni della compatibilità (oltre all'equivalenza) sono, siano T e S due tipi:

- i valori di S sono un **sottoinsieme** dei valori di T, ad esempio intervalli;
- i valori di S sono un **sottoinsieme di valori canonicamente corrispondenti** di T. Questo è tipico di tipi come, ad esempio, i tipi `float` e `int`, dove ogni `int` n ha un `float` canonicamente corrispondente $n.0$;
- i valori di S sono un **sottoinsieme di valori arbitrari corrispondenti** di T. In questo caso abbandoniamo il requisito di canonicità del punto precedente e assumiamo la presenza di una qualche trasformazione arbitraria che converta qualsiasi valore di S in un valore di T, ad esempio possiamo rendere compatibili `int` e `float` convertendo i `float` (ad esempio, tramite arrotondamento) in `int`;
- le **operazioni** sui valori di T sono possibili anche sui valori di S. Questo è l'esempio mostrato per il duck typing e il principio alla base di una **nozione di sottotipaggio** (spoiler);

Coercizione e Casting di Tipo

Gli ultimi tre punti sulle nozioni alternative di compatibilità dei tipi presuppongono l'esistenza di un meccanismo di **conversione dei tipi**, in grado di colmare le differenze tra valori di tipi diversi. Anche in questo caso, i linguaggi adottano due modi di effettuare queste conversioni.

Chiamiamo **coercizione di tipo** l'applicazione **implicita** di una conversione di tipo canonica/arbitraria. Un esempio è una funzione `sum` che accetta i `float` e, se le passiamo gli `int`, il compilatore/interprete inserisce le conversioni necessarie in modo implicito, senza segnalare un errore di compatibilità. In entrambi i casi, le **conversioni** sono **sintattiche**, quando i tipi condividono la stessa rappresentazione in memoria (è il caso, ad esempio, degli intervalli, per i quali non si applica alcuna conversione), oppure avvengono tramite una **conversione canonica/arbitraria**, che trasforma la rappresentazione in memoria di un valore di un certo tipo in un valore di un altro, ad esempio gli interi in `float` (canonica) e viceversa (arbitraria).

Chiamiamo **casting di tipo** l'annotazione **esplicita** nel linguaggio di una conversione di tipo (e di valore), che applica una procedura di conversione indicata dall'utente. Il casting ha anche un valore di documentazione, rendendo staticamente esplicite le conversioni di tipo. Come esempio di type casting, C e Java adottano una sintassi minimalista `S s = ( S ) t`, mentre Rust ne fornisce una più verbosa `let s: S = t as S`.

Inferenza di Tipo

Il type checker di un linguaggio verifica che un programma rispetti le regole imposte dal sistema di tipi (in particolare, la compatibilità). Per eseguire i suoi controlli, il type checker deve determinare il tipo delle espressioni presenti nel programma, utilizzando le informazioni sui tipi che il programmatore ha inserito nel programma.

Concretamente, il type checker determina il tipo delle espressioni visitando l'albero di parsing del programma, partendo dalle sue foglie (variabili e costanti di cui si conosce il tipo), scendendo fino alla radice e **calcolando il tipo delle espressioni dalle informazioni accumulate lungo il suo percorso** (ad esempio, il sistema di tipi potrebbe stabilire che + è un operatore che, applicato a due espressioni di tipo `int`, dà luogo a un'espressione di tipo `int`).

Sapendo che il type checker può **inferire** alcune informazioni da una quantità ridotta di annotazioni di tipo, i linguaggi possono risparmiare al programmatore il compito di annotare tutte le espressioni. **L'inferenza di tipo** è il processo di attribuzione dei tipi alle espressioni, senza annotazioni esplicite dei tipi.

Inferenza di Tipo

A volte, l'algoritmo di inferenza non è in grado di inferire direttamente il tipo di un'espressione, ma ha bisogno di mantenerne il tipo “aperto”, procedere con altre parti del programma e “tornare” su quell'espressione in un secondo momento; naturalmente, se ha raccolto tutte le informazioni disponibili e non è ancora in grado di fissare il tipo dell'espressione considerata, l'algoritmo “rinuncia” e segnala all'utente la necessità di ulteriori informazioni.

Tecnicamente, mantenere “aperto” il tipo dell'espressione significa assegnare a quest'ultima una variabile di tipo che, procedendo con l'esplorazione degli alberi di parsing, arricchisce di vincoli (ad esempio, potremmo incontrare un'operazione + applicata su di essa, che limita la gamma dei tipi possibili solo a quelli che la supportano). La procedura che esegue questo controllo sui vincoli è una strategia di risoluzione della programmazione logica nota come **algoritmo di unificazione**.

C non fornisce un supporto rilevante per l'inferenza di tipo. Java e Rust ne forniscono forme semplici. I linguaggi della famiglia ML, basati sul sistema di tipi Hindley-Milner [1,2], forniscono un supporto più completo per l'inferenza di tipo.

[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.

Algoritmo di Unificazione

```
unify( C ){
  if C == ∅ then []
  else
    let { S = T } ∪ C' = C
    if S == T
      unify( C' )
    else if isVar(S) ∧ S ∉ FV( T )
      unify( [ S↦T ]C' ) ∘ [ S↦T ]
    else if isVar(T) ∧ T ∉ FV( S )
      unify( [ T↦S ]C' ) ∘ [ T↦S ]
    else if S == S1 → S2 && T == T1 → T2
      unify( C' ∪ { S1 = T1, S2 = T2 } )
    else
      fail
```

X non appare libera in T (no definizioni ricorsive)

C è l'insieme delle equazioni di tipo che derivano dall'analisi del codice di un programma, ad esempio, l'espressione X=5 produce l'equazione X=Nat

con [X↦T] indichiamo "**rimpiazzare X con T**",

e.g., [X↦T](X = Nat) = T = Nat,

[X↦T](X → X) = T → T e

[X↦T]{ a, b, ... } = { a[X↦T], b[X↦T], ... }

A ∘ B è un operatore di composizione di sostituzione del tipo

A ∘ B = [X↦A(T) per ogni (X↦T) in B e

X↦T per ogni (X↦T) ∈ A con X ∉ dom(B)]

dove A(T) = T se T ∉ dom(A) e A(X → Y) = A(X) → A(Y)

[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.

Algoritmo di Unificazione

unify(C){
  if C == ∅ **then** []
  else
    let { S = T } ∪ C' = C
(1) **if** S == T
      unify(C')
(2) **else if** isVar(S) ∧ S ∉ *FV*(T)
      unify([S↦T]C') ∘ [S↦T]
(3) **else** if *isVar*(T) ∧ T ∉ *FV*(S)
      unify([T↦S]C') ∘ [T↦S]
(4) **else if** S == $S_1 \rightarrow S_2$ && T == $T_1 \rightarrow T_2$
      unify(C' ∪ { S_1 = T_1, S_2 = T_2 })
    else
(5)   **fail**

C = { X→X→INT = INT→Y, X→X = Y }

unify(C), **let** { X→X = Y } ∪ C' con C' = { X→X→INT = INT→Y }

(3) *isVar*(Y) ∧ Y ∉ **FV**(X→X) ⇒ unify([Y↦X→X]C')∘[Y↦X→X]

unify({ Y→INT = INT→Y } ∪ C'') ∘ [Y↦X→X] con C'' = {}

(4) S == Y→INT e T == INT→Y

unify({} ∪ { INT = Y, Y = INT }) ∘ [Y↦X→X]

let { INT = Y } ∪ C''' con C''' = { Y = INT }

(3) isVar(Y) ∧ Y ∉ **FV**(INT) ⇒

unify([INT↦Y]C''') ∘ [INT↦Y] ∘ [Y↦X→X]

unify({ INT = INT } ∪ C'''') ∘ [INT↦Y] ∘ [Y↦X→X]

(1) unify({}) ⇒ [] ∘ [INT↦Y] ∘ [Y↦X→X]

[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.

Algoritmo di Unificazione

$A \circ B = [$

$- Z \mapsto A(T) \; \forall \; (Z \mapsto T) \in B,$

$- Z \mapsto T \; \forall \; (Z \mapsto T) \in A \text{ con } Z \notin dom(B)$

$]$

dove $A(T) = T$ se $T \notin dom(A)$

$A(Z \rightarrow W) = A(Z) \rightarrow A(W)$

$\Rightarrow([] \circ [INT \mapsto Y]) \circ [Y \mapsto X \rightarrow X]$

$\Rightarrow [INT \mapsto Y, Y \mapsto X \rightarrow X]$

$C = \{ X \rightarrow X \rightarrow INT = INT \rightarrow Y, X \rightarrow X = Y \} [INT \mapsto Y, Y \mapsto X \rightarrow X]$

$\Rightarrow \{ Y \rightarrow INT = INT \rightarrow Y, Y = Y \} [INT \mapsto Y, Y \mapsto X \rightarrow X]$

$\Rightarrow \{ INT \rightarrow INT = INT \rightarrow INT, INT = INT \}$ ✅

[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.