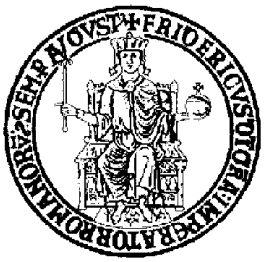
Università degli Studi di Napoli
Federico II



Dipartimento di Ingegneria Elettrica e delle Tecnologie dell'Informazione
Corso di Laurea Triennale in Informatica

Elaborato di Laurea

Design e Sviluppo del linguaggio di programmazione "Basalt"

Relatore:
Ch.mo Prof. Faella Marco

Candidato:
De Rosa Francesco
Matr. N86004379

Anno Accademico
2024/2025

Indice

1 Design del linguaggio

1.1 Introduzione

Basalt nasce con l'intenzione di offrire un'alternativa moderna a linguaggi di programmazione consolidati come C e C++. Nonostante questi ultimi siano ancora ben lontani dall'essere considerati obsoleti, è innegabile che comincino a mostrare i segni del tempo se paragonati con linguaggi moderni quali Go, Rust, Zig, Odin o Carbon.

L'obiettivo di Basalt è quello di essere semplice e minimale, facile da imparare, rimanendo al tempo stesso un linguaggio di basso livello e pertanto con gestione manuale della memoria. Basalt pone l'ergonomia al centro di ogni scelta di design, cercando di ridurre al minimo il tempo speso dal programmatore a correggere errori banali o scrivere codice ripetitivo.

Basalt, così come i sopra citati Go, Rust, Zig, Odin o Carbon, i quali saranno usati come termini di paragone per tutto il resto del capitolo, non adotta il paradigma ad oggetti. La nuova tendenza nei linguaggi di programmazione sembra essere di abbandonare il paradigma ad oggetti puro, offrendone una versione fortemente rivisitata o addirittura eliminandolo del tutto. Nel caso di Basalt, il ruolo operativamente ricoperto dalle interfacce è stato preso dalle union (sum-types), le quali possono essere usate per offrire funzionalità simil-polimorfiche mediante sostanziali integrazioni con le altre features del linguaggio.

Basalt, a differenza del C, è provvisto di un sistema dei tipi più avanzato, che permette di passare array statici o dinamici alle funzioni senza perdere informazioni riguardanti la dimensione di questi ultimi.

Basalt offre oltretutto pieno supporto alla programmazione generica, implementata mediante reificazione a tempo di compilazione (così come C++), e non mediante erasure, come ad esempio Java o Kotlin. Il supporto alla programmazione generica è stata una tra le prime features ad essere state implementate ed ha guidato molte delle scelte di design del linguaggio.

1.1.1 Confronto con altri linguaggi

Di seguito è stata presentata una tabella comparativa che mette a confronto Basalt con C, C++, Go e Java, evidenziando le caratteristiche comuni tra questi linguaggi e Basalt.

Si tenga presente che per molte delle feature elencate, sarebbe stato possibile considerare i loro corrispettivi inversi. Ad esempio, per la feature "gestione manuale della memoria", si sarebbe potuto considerare anche la feature "gestione automatica della memoria". L'obbiettivo di questa tabella comparativa è dunque non quello di dipingere Basalt come un linguaggio provvisto di ogni feature, ma semplicemente di considerare ogni feature di Basalt e verificare in quale dei linguaggi scelti come termine di paragone essa sia presente.

Features	***Basalt***	***C***	***C++***	***Go***	***Java***
gestione manuale della memoria	✓	✓	✓		
programmazione generica	✓		✓	✓	✓
union/variant	✓	✓	✓		✓
introspezione e riflessione				✓	✓
zero-cost-abstractions	✓	✓	✓		
compilato nativamente in linguaggio macchina	✓	✓	✓	✓	
indipendente da runtime-enviroment di qualsiasi tipo	✓	✓	✓		
non richiede header-files ed è invece basato su package/moduli	✓			✓	✓
non richiede che la definizione di un simbolo ne preceda l'utilizzo	✓			✓	✓
overloading di funzioni/metodi	✓		✓		✓
metaprogrammazione con supporto per annotazioni e decorator					✓
assenza di allocazioni di memoria nascoste o implicite	✓	✓	✓		
framework di unit-testing integrato nel linguaggio				✓	
Score:	10/13	6/13	8/13	6/13	7/13

Tabella 1: Confronto tra Basalt e altri linguaggi di programmazione

1.1.2 Struttura di un programma Basalt

Come precedentemente menzionato, Basalt si discosta dall'utilizzo di header files tipico di C e C++, optando invece per un sistema di gestione dei pacchetti simile a quello adottato da Java. In particolare, il sistema dei pacchetti di Basalt prevede che all'interno di un file appartenente ad un dato pacchetto, sia visibile il contenuto di tutti gli altri file dello stesso pacchetto, assieme al contenuto dei package importati.

Ogni file sorgente contenente codice Basalt deve possedere un'intestazione composta dalla dichiarazione del package corrente, ovvero il package a cui il file appartiene, e da una lista di package importati dal file, necessari al suo funzionamento.

Così come C, C++, Zig, Rust, Go, Jai, Odin e molti altri, il flusso di esecuzione parte da una chiamata fittizia ad una funzione speciale detta entry-point del programma. Così come da convenzione, tale funzione prende il nome di "main". Tale funzione deve necessariamente essere in un package di nome "main".

```
package main;

import console;

func main() {
    println("Hello, World!");
}
```

In maniera analoga a quanto è possibile vedere in Java, in Basalt importare un package non è una precondizione necessaria per l'utilizzo delle funzioni di tale package. É infatti possibile utilizzare la funzione `println` anche senza importare il package `console`, semplicemente riferendosi a tale funzione con il suo nome completo:

```
package main;

func main() {
    console::println("Hello, World!");
}
```

1.1.3 Indipendenza dall'ordine di definizione

Così come in Java, Rust e Go, e a differenza di C e C++, Basalt prevede che ogni definizione possa essere spostata in qualunque punto di un file sorgente o addirittura migrata in un altro file sorgente dello stesso package senza compromettere la correttezza del programma. In altri termini, in Basalt ogni definizione è accessibile non solo dalle definzioni che la succedono ma anche da quelle che la precedono.

L'indipendenza dall'ordine di definizione in un linguaggio di programmazione semplifica notevolmente il refactoring e l'utilizzo del codice. Il programmatore può riorganizzare e ottimizzare il codice senza dover preoccuparsi di errori di compilazione dovuti a riferimenti non ancora definiti. Questo favorisce una maggiore modularità e facilita il mantenimento del codice, poiché le modifiche possono essere apportate in modo più flessibile e incrementale. Inoltre, consente di migliorare la leggibilità del codice, organizzandolo in modo logico piuttosto che cronologico.

In un contesto di sviluppo collaborativo, questa caratteristica è particolarmente vantaggiosa, poiché diversi sviluppatori possono lavorare su parti diverse del codice senza doversi coordinare strettamente sull'ordine delle definizioni. Ciò riduce i conflitti di merge e accelera il processo di sviluppo. Anche l'aggiunta di nuove funzionalità o la correzione di bug risulta più semplice, in quanto le nuove definizioni possono essere inserite esattamente dove hanno più senso logico, senza dover riscrivere o spostare altre parti del codice esistente.

Ciò significa che il seguente codice è valido. Il compilatore è capace di risolvere correttamente il riferimento alla funzione `sum` anche se essa è definita dopo il suo primo utilizzo (ovvero la chiamata avvenuta nella funzione `main`).

```
package main;

func main() {
    var result : Int = sum(3, 5);
    console::print("The sum of 3 and 5 is: ");
    console::println(result);
}

func sum(a: Int, b: Int) -> Int {
    return a + b;
}
```

1.2 Variabili

Le variabili sono dei contenitori logici capaci di contenere dei valori decisi a tempo di esecuzione. Ci si potrà riferire al valore contenuto in un dato istante di tempo da una variabile o da una costante utilizzando il suo nome. Tramite un apposito costrutto detto assegnazione, è possibile riassegnare il valore di una variabile.

1.2.1 Dichiarazione di variabili

La dichiarazione di una variabile può avvenire con inizializzazione o senza, laddove un valore di inizializzazione sia mancante il valore di tale variabile sarà casuale. Ci si aspetta che in tale scenario un valore venga poi assegnato in un secondo momento. Qualunque sia la tipologia di dichiarazione scelta, essa deve essere introdotta dalla keyword var, seguita dal nome della variabile, dai due punti e dal tipo di tale variabile.

```
var x : Int = 6;
var y : Int;
```

1.2.2 Scoping di una variabile

Una variabile in Basalt esiste nello scope della funzione, del ciclo o più in generale del blocco di codice in cui è stata dichiarata. Ciò significa che una variabile dichiarata all'interno di un blocco di codice non sarà accessibile al di fuori di esso.

1.2.3 Shadowing

Con shadowing, si intende la possibilità di, all'interno di un blocco di codice innestato, dichiarare una variabile con un nome già usato in un blocco esterno, oscurandola. Numerosi linguaggi supportano lo shadowing delle variabili, ma Basalt non è tra questi. Si è ritenuto che tale funzionalità potesse portare a confusione e a codice di difficile comprensione, pertanto tentare di oscurare una variabile già definita in un blocco esterno causerà un errore a tempo di compilazione.

1.2.4 Deallocazione delle variabili

Al termine dell'esecuzione del blocco di codice corrente, l'area di memoria occupata dalle variabili dichiarate in esso verrà automaticamente deallocata. Nel caso in cui tali variabili abbiano per valore un indirizzo di memoria dinamica allocato in precedenza, la deallocazione di tale blocco **non** è automatica, e spetterà dunque al programmatore deallocare tale blocco di memoria manualmente.

1.3 Controllo del flusso di esecuzione

Con il termine "Control-Flow", o in italiano controllo del flusso di esecuzione, si intende l'iniseme dei costrutti che rendono l'esecuzione del codice non lineare, in particolare in Basalt essi sono cicli iterativi e branch condizionali.

1.3.1 Branch condizionali

Un branch condizionale, anche chiamato "if-statement", è un costrutto che consente di eseguire una porzione di codice solo se una certa conzione booleana è vera. Ad esmpio si consideri il seguente frammento di codice che illustra l'utilizzo di un if-statement.

```
var x : Int = math::random<Int>(0,10);
if (x % 2 == 0) {
    console::println("x is even");
}
```

In questo codice, l'istruzione `console::println("x is even")` sarà eseguita solo nel caso in cui il valore numerico intero attualmente contenuto nella variabile x sarà pari.
É possibile aggiungere un blocco di codice da eseguire nel caso in cui la condizione sia falsa utilizzando la keyword else. Ad esempio è possibile stampare del testo che informi l'utente del fatto che la variabile x contiene un valore dispari.

```
var x : Int = math::random<Int>(0,10);
if (x % 2 == 0) {
    console::println("x is even");
}
else {
    console::println("x is odd");
}
```

In Basalt l'indentazione non è rilevante, per cui, se lo si preferisce, è accettato (anche se sconsigliato) disporre la keyword else sulla stessa riga della chiusura della parentesi graffa relativa al blocco di codice da eseguire nel caso in cui la condizione sia vera.

1.3.2 Ciclo while

Il ciclo while è un costrutto utilizzato per ripetere una certa porzione di codice finchè una certa condizione booleana rimane vera. Il corpo del ciclo viene eseguito solo dopo aver controllato la condizione booleana. Si consideri ad esempio il seguente frammento di codice dove è presentato un ciclo while a scopo esemplificativo:

```
var i : Int = 0;
while (i < 10) {
    console.println(x);
    i = i + 1;
}
```

L'esecuzione di tale ciclo comporta la stampa in console dei numeri da 0 a 9. Più in generale, si può dire che un ciclo while è composto da condizione e corpo, e che la sua esecuzione avviene secondo il seguente diagramma di flusso (flow-chart).

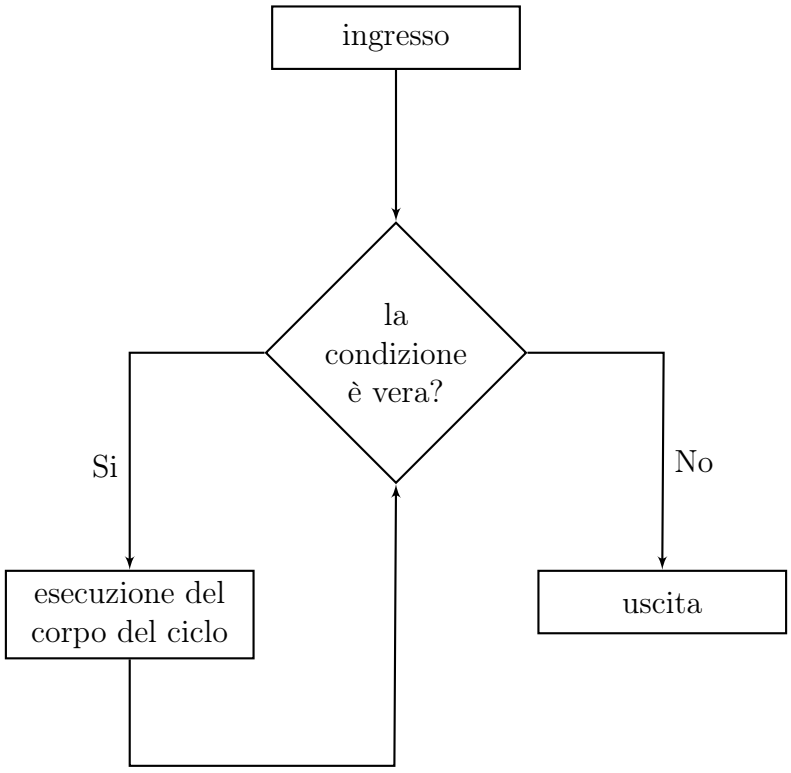


Figura 1: Diagramma di flusso del ciclo while

1.3.3 Ciclo until

Il ciclo until è un costrutto utilizzato per ripetere una certa porzione di codice finchè una certa condizione booleana rimane falsa. Il corpo del ciclo viene eseguito prima di aver controllato la condizione booleana. Si consideri ad esempio il seguente frammento di codice dove è presentato un ciclo until a scopo esemplificativo:

```
var i : Int = 0;
until (i > 10) {
    console.println(x);
    i = i + 1;
}
```

L'esecuzione di tale ciclo comporta la stampa in console dei numeri da 0 a 10. Così come per il ciclo while, si può dire che un ciclo until è composto da condizione e corpo, e che la sua esecuzione avviene secondo il seguente diagramma di flusso (flow-chart).

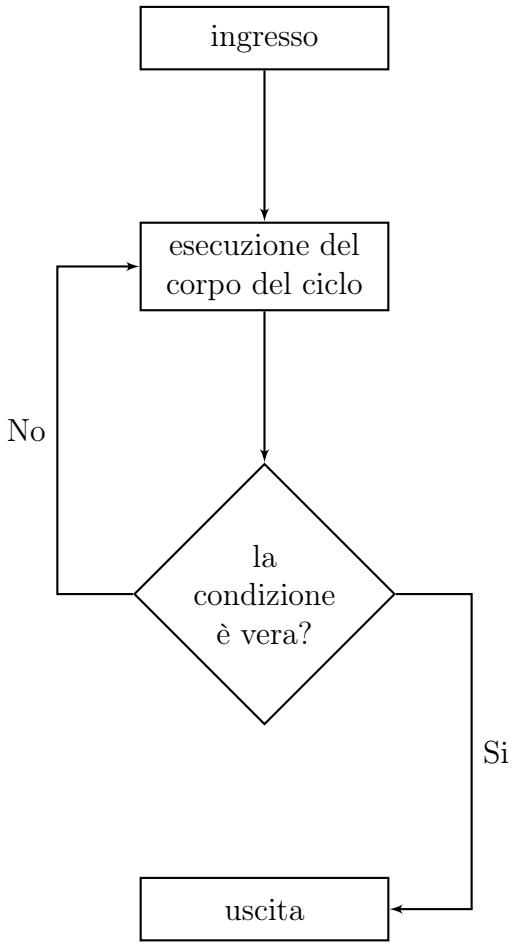


Figura 2: Diagramma del ciclo until

1.3.4 Break e continue

La keyword `break` consente di provocare l'interruzione anticipata da un ciclo. Essa è pensata per essere utilizzata assieme ad un branch condizionale che monitori una qualche condizione eccezionale che richiede l'interruzione immediata del ciclo.
Ad esempio, si analizzi il seguente frammento di codice che illustra un ciclo while:

```
while (true) {
    var x = math.random<Int>(-5,5);
    if (x % 3 == 0) {
        break;
    }
    console.println(x);
}
```

Tale ciclo presenta una condizione da controllare prima della stampa in console, ovvero la non divisibilità per 3 del valore contenuto nella variabile x. Qualora tale condizione si verificasse si uscirebbe immediatamente dal ciclo, altrimenti si procederebbe con la stampa in console del valore di x.

La keyword `continue`, similmente alla keyword break, consente di alterare il flusso di esecuzione di un ciclo. Anzichè provocarne l'interruzione anticipata, essa consente di saltare l'esecuzione del codice rimanente all'interno del corpo e passare direttamente alla successiva iterazione. Ad esempio, si consideri il seguente frammento di codice:

```
var x : Int = 0;
while (x < 10) {
    if (x % 3 == 0) {
        continue;
    }
    console.println(x);
    x = x + 1;
}
```

L'esecuzione di questo ciclo avrà come effetto la stampa in console dei numeri da 0 a 9 che non sono divisibili per 3, in quanto ad ogni iterazione dove x avrà valore divisibile per 3, la stampa in console sarà saltata e si proseguirà all'iterazione seguente.

1.4 Tipi primitivi

Il sistema dei tipi, spesso più comunemente chiamato Typesystem, è un insieme di regole che definiscono il comportamento e le operazioni consentite su tipi di dati. Questo sistema è fondamentale per garantire la correttezza e la sicurezza del codice. In Basalt, tale sistema prevede un insieme di tipi detti tipi primitivi, i quali esistono nativamente nel linguaggio, e permette all'utente di definire tipi personalizzati.

1.4.1 Tipi primitivi semplici

Con "tipi primitivi semplici", in Basalt, si intendono i seguenti tipi di dato:

IDENTIFICATIVO	***DESCRIZIONE***
`Int`	tipo di dato preposto alla rappresentazione dei numeri interi, rappresentato a 64 bit
`Float`	tipo di dato preposto alla rappresentazione dei numeri decimali frazionari, internamente analogo ad un double in C/C++
`Bool`	tipo di dato preposto alla rappresentazione di valori logici (booleani) di vero/falso
`Char`	tipo di dato preposto alla rappresentazione di un singolo carattere ascii 8 bit

Tabella 2: Tipi primitivi

In Basalt, variabili il cui tipo è un tipo primitivo semplice, vengono allocate su stack. Tutte le volte che si lavora con una variabile così dichiarata, si deve dunque assumere che essa si trovi sullo stack della funzione corrente (compresi gli argomenti delle funzioni).

1.4.2 Array

In Basalt, gli array sono dei blocchi di memoria contigua, capaci di contenere un numero noto a tempo di compilazione di oggetti dello stesso tipo.

Dato un tipo `T` ed una lunghezza N allora il tipo `[N]T` denoterà il tipo di un array contenente esattamente N oggetti di tipo `T`. Basalt conserva la lunghezza come parte del tipo, ciò implica che è possibile definire una funzione che prenda come parametro di input un array di cui sia specificata la lunghezza, a differenza del C dove invece si è obbligati a passare la lunghezza tramite l'utilizzo di un parametro ausiliario.

Basalt supporta array-literals sottoforma di tipo esplicito dell'array, seguito da una lista di valori separati da virgole e racchiusi tra parentesi graffe. Tale sintassi può essere usata per inizializzare un array in sede di dichiarazione come illustrato di seguito.

```
var array : [10]Int = [10]Int{0,1,2,3,4,5,6,7,8,9}
```

Così come in quasi tutti i linguaggi imperativi ad oggi usati, dato un array, si può accedere in lettura (e in scrittura qualora non sia costante) al suo ennesimo elemento usando la canonica sintassi storicamente introdotta dal C, che prevede di posporre all'espressione costituente l'array e racchiusa tra parentesi quadre, un'espressione il cui valore sia intero e che corrisponda alla posizione dell'elemento all'interno dell'array, assumendo un'indicizzazione che parte da zero.

In generale, un array occupa in memoria un numero di byte pari al prodotto della dimensione in byte di un singolo oggetto in esso conservato, moltiplicato per la lunghezza, ed è dunque privo di qualunque overhead dato che la dimensione è nota a tempo di compilazione e pertanto non viene conservata in memoria.

Un assignment tra array è possibile solo se hanno la stessa dimensione e se i tipi degli oggetti in essi conservati sono tali da consentire un ipotetico assegnamento cella a cella. Qualora tali requisiti siano soddisfatti allora l'assignment performerà una copia di tutti gli elementi dell'array sorgente nell'array destinazione.

1.4.3 Puntatori scalari

In Basalt, i puntatori scalari, più semplicemente detti puntatori, sono dei riferimenti ad un oggetto allocato in memoria, avente un certo tipo noto a tempo di compilazione.

Dato un qualunque tipo `T`, allora con `#T`, indichiamo il tipo dei puntatori a oggetti di tipo `T`. In Go, C e C++ il simbolo preposto a questo scopo è l'asterisco ("∗"), mentre in Jai il simbolo preposto a questo scopo è il carot ("^"). Il motivo per cui Basalt si discosta dagli altri linguaggi per quanto riguarda il simbolo usato per indicare un puntatore è che Basalt vuole cercare di non usare lo stesso simbolo in contesti troppo diversi fra loro. In particolare, dato che l'asterisco e il carot sono simboli già in uso in qualità di operatori binari, è sembrato più saggio scegliere un altro simbolo da dedicare allo scopo di indicare i puntatori.

In maniera conforme a quanto visto in C, C++, Go e molti altri linguaggi, dato un qualunque oggetto di tipo `T`, l'operatore unario prefisso `&` consente di estrarre l'indirizzo di memoria di tale valore. Tale indirizzo avrà tipo `#T` e sarà per tanto assegnabile ad un puntatore a Type come mostrato nel seguente esempio.

```
var number : Int = 6;
var ptr : #Int = &number;
```

Ad un puntatore è possibile assegnare un valore fittizio detto null per rappresentare il fatto che in quel momento il puntatore non sta puntando a un'area di memoria valida.

I puntatori possono riferirsi sia ad aree di memoria su stack sia su heap, ma per allocare memoria su heap sarà necessario chiamare manualmente funzioni di allocazione. Una volta allocata memoria, essa dovrà essere deallocata manualmente in quanto Basalt non possiede un garbage collector a differenza di Go e Java, e invece consente all'utente di gestire la memoria manualmente così come C, C++, Zig, Odin e Jai.

Nel package memory è possibile trovare una funzione malloc e una funzione free, preposte all'allocazione e alla deallocazione di memoria dinamica su heap, di seguito è riportato un esempio d'uso. Si tenga a mente che la sintassi con le parentesi angolari sarà analizzata con maggior dettaglio in seguito nella sezione dedicata ai generics.

```
var ptr : #Int = memory::malloc<Int>(6);
memory::free<Int>(ptr);\vspace{0.5cm}
```

1.4.4 Puntatori vettoriali

Contrapponendosi ai puntatori scalari vi sono poi i puntatori vettoriali. Un puntatore vettoriale è un puntatore ad una sequenza di oggetti contigui in memoria il cui tipo è noto a tempo di compilazione, ma la cui lunghezza è nota a tempo di esecuzione. Per semplicità è possibile chiamarli "slice" così come si fa in molti altri linguaggi.

I puntatori vettoriali sono internamente implementati come una coppia di un puntatore ed una dimensione. Dato un tipo T allora il tipo $T ne denoterà il puntatore vettoriale.

Un puntatore vettoriale in una macchina a 64bit occupa internamente 16 byte, di cui 8 sono dedicati a conservare un indirizzo di memoria ed altri 8 sono dedicati a conservare la lunghezza, ovvero il numero di celle contigue allocate a partire da tale indirizzo.

A differenza dei puntatori scalari, un puntatore vettoriale non può essere null, però può avere dimensione zero, che è infatti il comportamento standard per un puntatore vettoriale non ancora inizializzato. Questo consente di poter scrivere codice che lavora con puntatori vettoriali senza doversi assicurare ogni volta che il puntatore sia non nullo, ma semplicemente controllando di accedere sempre ad esso con indici strettamente minori della sua dimensione come è del resto naturale fare anche per gli array.

La sintassi per accedere all'i-esimo elemento di un puntatore vettoriale è del tutto uguale a quanto già visto per gli array, ovvero si pone alla destra del puntatore vettoriale, da cui si desidera leggere, un espressione di tipo intero, il cui valore numerico sarà interpretato come indice, racchiusa fra parentesi quadre.

Il seguente frammento di codice illustra come si può istanziare un blocco di memoria dinamica su heap e come lo si può gestire mediante un puntatore vettoriale a tale blocco. In particolare il seguente codice stampa il contenuto di ogni cella del blocco.

```
var i : Int = 0;
var slice : $Int = malloc<$Int>([5]Int{0, 1, 2, 3, 4});

while (i < slice.size){

    println(slice[i]);
    i = i + 1;
}
memory::free<$Int>(slice);
```

É possibile assegnare ad una variabile di tipo "puntatore vettoriale a T", un'espressione di tipo "puntatore scalare ad array di oggetti di tipo T" di qualsiasi dimensione. Ciò consente l'utilizzo del puntatore vettoriale come supertipo di tutti gli array. Tale assegnazione comporta un effetto simile a quello osservato quando si assegna l'indirizzo di un oggetto già istanziato a un puntatore utilizzando l'operatore di indirizzo `&`. In tal modo, entrambi i riferimenti puntano alla stessa area di memoria.

```
var array : [10]Int = [10]Int{0,1,2,3,4,5,6,7,8,9};
var slice : $Int = &array;
```

Dato che un puntatore vettoriale, così come un puntatore scalare, non conserva informazioni sufficenti a determinare se l'oggetto puntato si trovi su stack o su heap, e dato che Basalt si prefigge come obbiettivo quello di non effettuare allocazioni nascoste e invece di essere sempre trasparente riguardo alla gestione della memoria, ne consegue che non è possibile inserire nuovi elementi in un puntatore vettoriale o ridimensionarlo in qualsiasi altro modo.

Un'ipotetica implementazione di un array dinamico propriamente detto con possibilità di inserire e rimuovere elementi da esso potrebbe essere quella mostrata nel seguente frammento di codice. Si tenga a mente che tale frammento usa struct e generics, entrambi argomenti che saranno trattati in dettaglio nelle loro sezioni apposite.

```
package slicedemo;

struct Slice<T> {
    storage : $T;
    size : Int;
}

func append<T>(slice : Slice<T>, value : T){
    if (slice.size + 1 > slice.storage.length){
        var old : $T = slice.storage;
        var new_length = 2 * slice.storage.length;
        slice.storage = memory::malloc<$T>(new_length);
        memory::copy<T>(old, slice.storage);
        memory::free<$T>(old);
    }
    slice.storage[slice.size] = value;
    slice.size += 1;
}
```

1.4.5 Stringhe

La gestione delle stringhe nei linguaggi di basso livello è da sempre una sfida. In C, C++, Zig e Odin le stringhe non sono altro che puntatori ad aree di memoria contigue dove sono conservati dei caratteri. Tale è anche l'approccio di Basalt, dove le stringhe, indicate con String, sono implementate come puntatori vettoriali a carattere.

Per facilitare l'interoperabilità con C, esiste anche il tipo RawString che è invece implementato come un puntatore scalare a carattere, il quale, punta al primo carattere della sequenza che compone la stringa. In C infatti, una stringa altro non è che un puntatore al primo carattere che ne fa parte. Non avendo una dimensione, le stringhe in C devono essere marcate al termine da un carattere speciale '\0' che ne segnala la terminazione. Al fine di poter convertire agevolmente una String in una RawString, la quale può essere usata per interfacciarsi con C, allora in Basalt è comunque presente il carattere speciale '\0' al termine di ogni sequenza di caratteri conservata in ogni oggetto di tipo String anche se superfluo. Si analizzi dunque il seguente codice.

```
var str : String = "hello world!";
var cstr : RawString = str;
```

Nel frammento di codice appena mostrato si assegna il valore di una variabile di tipo String ad una variabile di tipo RawString. É possibile descrivere graficamente lo stato della memoria al termine dell'esecuzione di questo frammento di codice con un Memory-Layout-Diagram nel seguente modo:

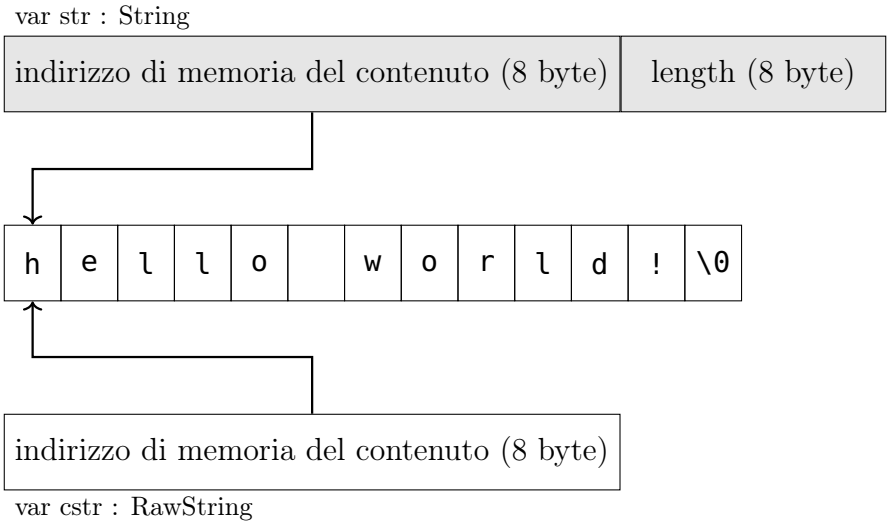


Figura 3: Memory layout dei tipi String e RawString

Qualunque string-literal, quindi anche "Hello, World!" nell'esempio di prima, viene implicitamente spostata nello scope globale così che dall'interno di una funzione si possa restituire una string-literal senza temere che alla fine della chiamata lo stack della funzione venga ripulito e che la stringa appena restituita venga sovrascritta o invalidata.

Questo meccanismo di gestione delle string-literals viene chiamato string-pooling, e l'area di memoria nello scope globale dedicata a contenere tutte le string-literal dell'intero programma viene detta string-pool. Questo meccansimo consente poi di non dover replicare le string-literal, infatti qualora una stessa string-literal apparisse più volte nel programma in scope diversi, sarebbe comunque utilizzato l'indirizzo della stessa unica string-literal nella pool per inizializzare variabili o per effetturare accessi in lettura.

Così come in Go e in Java, le stringhe sono immutabili, questo è un prerequisito essenziale al funzionamento dello string-pooling, dato che stringhe in scope diversi si riferiscono in realtà alle stesse sequenze di caratteri nella pool.

Per modificare una stringa, occorrerà dunque allocare una nuova area di memoria per ospitare i caratteri della stessa (su stack utilizzando un array di caratteri da castare successivamente a puntatore vettoriale o su heap utilizzando una funzione di allocazione e gestendo un puntatore vettoriale direttamente). Si procederà alla modifica e si assegnerà il puntatore vettoriale dell'area di memoria contenente la nuova sequenza di caratteri modificata alla stringa che si desiderava modificare. In Go accade qualcosa di sostanzialmente analogo. Segue un conciso esempio concreto di quanto appena detto.

```
var str : String = "some text";
var tmp : $Char = memory::malloc<Char>(str);

var i : Int = 0;
while (i < tmp.length){
    tmp[i] = uppercase_character(tmp[i]);
    i += 1;
}

str = tmp;
console::println(str);
memory::free<String>(str);
```

Si noti come in questo caso, dato che la stringa ora conserva un riferimento ad un'area di memoria dinamica, allocata su heap manualmente, si rende dunque necessario effettuare una deallocazione manuale al termine dell'utilizzo con la funzione free.

1.5 Struct

Le struct, abbreviazione di "structures" in inglese, rappresentano un fondamentale costrutto di molti linguaggi di programmazione, incluso Basalt. Una struct è difatti un tipo definito dall'utente, preposto alla modellazione di entità complesse, concettualmente rappresentabili come un aggregato di dati distinti.

In altri linguaggi, costrutti analoghi sono chiamati "records" o "product-types".

In Basalt, la definizione di una struct avviene utilizzando la parola chiave `struct` seguita dal nome della struct, che deve iniziare per maiuscola come ogni altro tipo nel linguaggio, e da una serie di campi all'interno di parentesi graffe.

Ogni campo deve essere nella forma <nome> : <tipo>, come mostrato di seguito:

```
struct Person {
    name : String;
    surname : String;
    occupation : String;
}
```

Una volta definita una struct, è possibile usare il suo nome nei contesti dove il linguaggio Basalt richiede un tipo, come ad esempio nella dichiarazione di una variabile, nei parametri delle funzioni o come tipo di un campo di un'altra struct.
Su ogni variabile il cui tipo è una struct, è possibile applicare l'operatore binario "." così come nella maggior parte dei linguaggi di programmazione, tale operatore consente di accedere ai campi di una specifica istanza di una struct.
Assumendo dunque di avere accesso alla definizione di Person dal precedente frammento di codice, sarà dunque lecito dichiarare variabili di tipo Person e accedere in lettura e scrittura ai loro campi utilizzando l'operatore "." avendo come operatore sinistro un oggetto di tipo Person e come operatore destro il nome di uno specifico campo.

```
var john : Person;

john.name = "John";
john.surname = "Doe";
john.occupation = "Programmer";
```

1.5.1 Puntatori a struct

In C e in C++, per accedere ai campi di un oggetto dato un puntatore a tale oggetto, occorre o dereferenziare il puntatore, per poi utilizzare l'operatore "." sull'oggetto così ottenuto, oppure usare l'operatore apposito "->" funzionalmente analogo.

In Basalt, così come in Go, è possibile usare l'operatore "." direttamente sul puntatore per accedere ai campi dell'oggetto puntato. Tale sintassi non porta ambiguità dato che non vi sono altri significati per l'operatore "." applicato ad un puntatore.

Consideriamo infatti il seguente frammento di codice, dove vengono definite diverse variabili, e alcune delle quali sono puntatori. In questo esempio, sarà fatto riferimento alla definizione per la struct Person data nella pagina precedente.

```
var person : Person;
var person_ptr : #Person = &person;
var person_ptr_ptr : ##Person = &person_ptr;
```

Allora si potrà accedere al campo "name" della variabile "person" posponendo ".name" a una qualunque di queste tre variabili.

Go è stato il primo linguaggio ad introdurre un meccanismo del genere, seppur in una forma più limitata dove è possibile accedere al campo dell'oggetto puntato solo da un puntatore che vi punti direttamente, e non consentendolo invece nei casi dove vi sono puntatori a puntatori, o più in generale, due o più livelli di indirezione.

Tale sovraccarico della semantica dell'operatore ".", consente di ridurre al minimo le modifiche da effettuare ad un blocco di codice funzionante qualora si voglia decidere di cambiare il tipo di una delle variabili che esso utilizza rendendola un puntatore invece che un oggetto locale. Dunque la scelta di estendere l'utilizzo di tale operatore in tal modo è stata fatta per facilitare refactoring del codice.

Ciò è particolarmente vero nei casi in cui una funzione utilizza già un puntatore per accedere in lettura e scrittura ai campi di un oggetto e ci si accorge in un secondo momento che tale funzione ha bisogno di eventualmente riassegnare un nuovo valore all'oggetto stesso. In tal caso, l'unica modifica da apportare alla funzione sarà cambiare il tipo dell'argomento in questione e rendendolo un puntatore a puntatore, mantenendo il resto del codice intatto. Chiunque abbia programmato C abbastanza a lungo si potrà facilmente rendere conto che tale scenario è molto comune e pertanto facilitare la risoluzione di un problema del genere è qualcosa di cui il programmatore medio può beneficiare in modo concreto e tangibile.

1.5.2 Struct ricorsive

Le variabili al momento della creazione, sia su stack che su heap, devono avere una dimensione in bytes nota a tempo di compilazione. Tale dimensione, per le variabili il cui tipo è una struct, è ottenuta calcolando la somma delle dimensioni dei field, i cui tipi possono potenzialmente essere anch'essi struct.

Date queste premesse, è chiaro che definizioni ricorsive come la definizione seguente, sono errate e portano ad un errore a tempo di compilazione.

```
struct Recursive {

    // Ricorsione diretta -> Errore
    recursive : Recursive;
}
```

Il compilatore Basalt, non potrebbe calcolare la dimensione in byte di un ipotetico oggetto il cui tipo è Recursive e di conseguenza, causa un errore di compilazione.

Basalt riesce ad identificare questo errore esplorando il grafo orientato che le definizioni di struct implicitamente descrivono ed implementa un controllo di aciclicità su di esso.

Basalt in tale controllo si limita ad esplorare gli archi relativi a tipi semplici e array, e invece non esplora archi relativi a puntatori scalari e vettoriali. Questo perchè un puntatore, vettoriale o scalare, ha sempre dimensione nota. Ne consegue che questa definizione alternativa della struct Recursive è invece corretta e perfettamente valida.

```
struct Recursive {

    // Ricorsione indiretta -> Corretto
    recursive_ptr : #Recursive;
    recursive_slice : $Recursive;
}
```

1.6 Union

Le union sono un costrutto che consente al programmatore di definire un tipo di dato la cui rappresentazione interna può variare nell'ambito di un numero finito di opzioni mutuamente esclusive e note a priori.

Le union in Basalt non sono implementate come in C, e sono invece più simili ad i "sum-types" presenti in molti linguaggi funzionali come Haskell, Idris o ML.

In Basalt, la definizione di una union avviene utilizzando la parola chiave `union` seguita dal nome della union, che deve iniziare per maiuscola come ogni altro tipo nel linguaggio, dal simbolo uguale, e da una serie di tipi separati da "|" (pipe).

```
union Number = Int | Float
```

Non è necessario definire una union dandole un nome, è infatti possibile utilizzare union anonime, ovvero union definite su una singola riga direttamente al momento dell'utilizzo.

La sintassi per fare ciò, prevede semplicemente di utilizzare una serie di tipi separati da "|" in tutti i contesti in cui il type-system richiede l'utilizzo di un tipo. In automatico tale entità verrà interpetata come union-anonima.

```
var named_union_example : Number = 3.14;
var inline_union_example : Int | Float = 7;
```

1.6.1 Union ricorsive

Una union, così come una struct, deve avere una dimensione in byte nota a tempo di compilazione, e tale dimensione è funzione delle dimensioni dei tipi a partire dai quali essa è definita. Analogamente a quanto visto per le struct dunque, la seguente definizione non è valida in quanto Basalt non è in grado di calcolare la dimensione di una ipotetica variabile di tipo Recursive.

```
union Recursive = Int | Recursive
```

Per gli stessi motivi per cui ciò era valido per le struct, è però possibile definire union ricorsive con la ricorsione indiretta, ovvero usando puntatori (vettoriali e scalari).

```
union Recursive = #Recursive | $Recursive
```

1.6.2 Memory-layout di una union

Come già detto nel paragrafo precedente, è stato detto che la dimensione in Byte occupata da una union, è calcolata in funzione della dimensione del tipo con la dimensione più grande tra quelli a partire dalla quale essa è stata definita.

Una union è internamente rappresentata come due blocchi di byte adiacenti in memoria, il primo, di 8 byte, è detto header, ed è usato per contenere metadati necessari al corretto funzionamento dell'operatore is, mentre il secondo è detto payload, e contiene la rappresentazione in byte del valore rappresentato a tempo di esecuzione dalla union.

Figura 4: Memory layout dei tipi String e RawString

Definiamo dimensione netta di un tipo la dimensione del suo payload, se esso è una union, o la sua dimensione complessiva in byte se esso è un tipo di altra natura.

La dimensione in byte del payload di una union, ovvero la sua dimensione netta, è pari alla dimensione netta del tipo con dimensione netta maggiore tra quelli a partire dai quali la union è stata definita.

Per union definite a partire da altre union dunque, gli overhead dati dagli header non sono cumulativi. Gli 8 byte dedicati all'header sono usati per conservare l'indirizzo in memoria a cui sono conservate le type informations relative al tipo di volta in volta contenuto all'interno della union. In sede di assignment, che è l'unica occasione in cui il tipo contenuto possa cambiare, tale puntatore viene eventualmente aggiornato.

L'assignment ad una union quindi è in realtà una coppia di due operazioni, la prima è la scrittura dei byte all'interno del payload (nel caso in cui il tipo del valore assegnato sia una union, saranno copiati solo i byte del payload), la seconda è la scrittura dei byte relativi all'header con l'indirizzo, staticamente noto, delle type-informations del tipo che si è andati ad assegnare (nel caso in cui tipo del valore assegnato sia una union, saranno copiati i byte del suo header all'interno dell'header della union destinazione).

Qualcosa di funzionalmente analogo a quanto descritto fin ora, sono le std::variant introdotte nella libreria standard C++ a partire dallo standard C++17. Esse non sono parte del core language, e sono invece definite usando la metaprogrammazione C++.

1.6.3 Operatore `is`

Per conoscere il tipo effettivo rappresentato in un certo momento dell'esecuzione del programma da un oggetto il cui tipo è una union, si può utilizzare l'operatore is, il quale si comporta in modo analogo ad instanceof in java o all'omonimo operatore is in C#, ovvero restituisce true se e solo se il tipo concreto dell'oggetto fornito come operando sinistro è assegnabile al tipo fornito come operando destro.

```
var num : Int | Float = 6;

if (num is Int) {
    console::println("num is an integer");
}
else {
    console::println("num is a float");
}
```

1.6.4 Operatore `as`

Per poter accedere al valore internamente contenuto da una variabile il cui tipo è una union è possibile usare l'operatore `as`, operatore binario il cui operando sinistro è un'espressione il cui tipo è una union, mentre l'operando destro è un tipo che si desidera estrarre dalla union.

L'operatore as, è utilizzabile solo su un tipo che sarebbe teoricamente assegnabile all'espressione sulla quale esso viene usato, pena un errore a tempo di compilazione.

Se lo si usa su espressioni che contengono un tipo diverso, esso non fallisce a tempo di esecuzione, ma si limita a fornire valori indefiniti corrispondenti all'interpretazione dei byte del contenuto reale della union come se essi fossero invece del tipo richiesto.

L'uso dell'operatore as è consigliato solo all'interno di dei branch condizionali, o dopo degli assert, la cui condizione assicura che la union contenga effettivamente il tipo che il programmatore si aspetti a tempo di esecuzione.

L'operatore as fornsice un vero e proprio riferimento utilizzabile non solo in lettura ma anche in scrittura, è analogo al reinterpret-cast di C++ ma, se usato in condizioni in cui l'operatore is con gli stessi operandi avesse valore true, allora il suo buon funzionamento è sempre garantito.

1.7 Generics

Con generics ci si riferisce a parametri formali di tipo applicabili a definizioni di tipi e funzioni all'interno del linguaggio di programmazione Basalt.

Tali definizioni diventano così parametriche, vengono dunque sottoposte a un type-checking ridotto e vengono utilizzate come dei template per generare definizioni concrete (non-parametriche) al momento del loro utilizzo, istanziandole con i valori concreti di tali parametri di tipo.

Tale approccio all'implementazione dei generics è detto "reificazione" ed è usato da linguaggi come ad esempio `C++`, a tempo di compilazione, e da `C#`, a tempo di esecuzione. Al contrario, linguaggi come Java e Kotlin usano un approccio detto "erasure".

1.7.1 Struct generiche

In Basalt, le struct sono parametrizzabili mediante l'utilizzo dei generics. La sintassi per definire una struct generica prevede una lista di identificatori di tipo separati da virgole e racchiusi in parentesi angolari alla destra del nome della struct. Di seguito viene riportata la definizione di una linked list doppiamente puntata e parametrica sul tipo di dato conservato in ogni nodo.

```
struct List<T> {
    size : Int;
    head : #Node<T>;
    tail : #Node<T>;
}

struct Node<T> {
    item : T;
    next : #Node<T>;
    prev : #Node<T>;
}
```

1.7.2 Union generiche

Così come le struct, anche le union possono essere generiche (se non anonime), ovvero possono avere parametri formali di tipo. Anche nel caso delle union la loro implementazione concreta consiste nella reificazione a tempo di compilazione.

Un caso particolarmente indicativo dell'utilità di questo costrutto è ad esempio una ipotetica union Collection, generica con parametro di tipo T, definita a partire da una serie di tipi definiti come struct, i quali implementano varie strutture dati.

```
union Collection<T> = LinkedList<T> | HashTable<T> | Tree<T>
```

1.7.3 Funzioni generiche

Come detto in precedenza, le funzioni in Basalt possono essere generiche. La definizione di una funzione generica prevede la presenza di una lista non vuota di parametri formali di tipo, separati da virgole e racchiusi tra parentesi angolari, che precede la lista di argomenti della funzione.

È possibile definire una funzione generica che restituisca il massimo tra due valori il cui tipo è specificato al momento della chiamata.

```
func max<T>(first : T, second : T) -> T {
    if (first > second) {
        return first;
    }
    else {
        return second;
    }
}
```

Tale definizione sarà istanziata all'occorrenza e sarà possibile istanziare tale funzione solo per tipi confrontabili con l'operatore '>'. Istanziare tale funzione con tipi non confrontabili genererà un errore a tempo di compilazione.

In sede di chiamata a funzione, è possibile specificare dei parametri attuali di tipo per la funzione stessa dopo il nome e prima dell'elenco degli argomenti, elencandoli separati da virgole e racchiusi tra parentesi angolari.

Ad esempio per la funzione add è possibile usare sia `Int`, che `Float`.

```
var x : Int = max<Int>(3, 5);
var y : Float = max<Float>(3.14, 5.17);
```

Così come è lecito aspettarsi, è possibile utilizzare union, eventualmente anche anonime, come parametri attuali di tipo per funzioni generiche.

```
func f<T>(x : T, y : T) {
    /* no-op */
}
```

```
f<Int|String>(3, "Hello");
```

1.7.4 Type-inferece

Per funzioni generiche è possibile non specificare espressamente dei parametri attuali di tipo e lasciare che sia Basalt a dedurli dal contesto. L'operazione di deduzione dei parametri attuali di tipo a partire dal contesto è detta type-inference.

In sede di chiamata a funzione, se tale funzione è generica e non vengono specificati i tipi dei parametri attuali, Basalt analizzerà uno ad uno gli argomenti forniti e cercherà di risolvere i vincoli di assegnabilità.

Per ogni argomento, Basalt analizzerà il tipo dichiarato in sede di definizione della funzione e lo comparerà con il tipo dell'argomento. Se il tipo dell'argomento è generico, e tale generico viene usato una sola volta, allora Basalt assumerà che il parametro attuale di tipo corrispondente sia proprio il tipo dichiarato dell'argomento.

Qualora il tipo dell'argomento sia un generico ed è già stato usato in precedenza, Basalt metterà in discussione la sua iniziale assunzione, qualora sia possibile, deducendo eventualmente un tipo più generale capace di ricevere assegnamenti di espressioni sia del tipo corrispondente alla sua precedente assunzione, sia di tipo corrispondente al tipo concreto dell'argomento.

In certi casi, è impossibile mettere in disussione l'assuzione fatta dalla type-inference, ad esempio, si consideri la seguente funzione generica:

```
struct Wrapper<T> {
    var value : T;
}

func f<T>(x : Wrapper<T>, y : T) {
    /* no-op */
}
```

Allora, la chiamata a funzione `f` con argomenti `Wrapper<Int>` e `Float` non sarà risolta su questo overload in quanto la type-inference prima assumerà che `T` sia `Int`, successivamente però, tenterà di aggiornare la sua assunzione precedente e dedurrà che `T` debba essere `Int|Float`, ma ciò renderà impossibile l'assegnamento di `Wrapper<Int>` a `Wrapper<T>`.

Si tenga presente che in Basalt è sempre possibile trovare un tipo `T` tale da ricevere assegnamenti da parte di espressioni di due tipi concreti noti `X1` ed `X2`, e tale tipo è la union anonima `X1 | X2`. Ad esempio sarà possibile chiamare la seguente implementazione della funzione `g` con argomenti arbitrari e la type-inference risolverà il tipo `T` come la union dei tipi concreti degli argomenti della chiamata.

```
func g<T>(x : T, y : T) {  /* no-op */ }
```

Il seguente programma Basalt illustra l'utilizzo della type-inference per funzioni generiche:

```
package type_inference_demo;

struct Wrapper<T> {
    value : T;
}

func f<T>(w : Wrapper<T>, t : T) {
    /* no-op */
}

func g<T>(x : T, y : T) {
    /* no-op */
}

func h<T>(x : T, y : Wrapper<T>) {
    /* no-op */
}

func demo() {
    var w1 : Wrapper<Float>;
    var w2 : Wrapper<Float|String>;

    f(w1, 3.2);
    // T = Float

    f(w2, "Hello");
    // T = Float|String

    // f(w1, "Hello");
    // T = Float|String
    // ERROR: Wrapper<Float|String> = Wrapper<Float> not allowed

    g(3.2, 3.2);
    // T = Float

    g(3.2, "Hello");
    // T = Float|String

    h(3.2, w2);
    // T = Float|String
}
```

1.8 Funzioni

Una funzione è un blocco di codice riutilizzabile molteplici volte la cui esecuzione può venire influenzata dal valore di eventuali parametri, qualora presenti, detti argomenti. Ogni argomento ha un tipo ed un nome, con il quale è possibile riferirsi ad esso.

Una funzione può "restituire" un valore al chiamante, ed il tipo di tale valore di ritorno è noto a tempo di compilazione qualora presente. È anche possibile che una funzione non restituisca nulla al chiamante, in tal caso si parla di procedura o di routine.

La definizione di una funzione in Basalt avviene utilizzando la parola chiave func, seguita dal nome della funzione, da un eventuale elenco non vuoto di parametri formali di tipo separati da virgole e racchiusi tra parentesi angolari, da un elenco eventualmente anche vuoto di argomenti, separati da virgole e racchiusi in parentesi tonde, da un eventuale tipo di ritorno preceduto dal simbolo "->" e infine da un blocco di codice delimitato da parentesi graffe chiamato corpo della funzione.

Di seguito è riportato un esempio di definizione di una funzione "max" che accetta due argomenti di tipo Int e restituisce un Int corrispondente al valore maggiore tra loro.

```
func max(first : Int, second : Int) -> Int {
    if (first > second) {
        return first;
    }

    else {
        return second;
    }
}
```

È possibile invocare questa funzione passando una qualunque coppia di valori interi, e più in generale è possibile invocare una funzione con una lista di valori i cui tipi siano compatibili per assegnazione ai tipi degli argomenti di tale funzione. La sintassi della chiamata a funzione in Basalt è equivalente a quanto si può vedere in linguaggi come C, C++ e Go, e consta del nome della funzione, seguito da una eventuale lista di parametri attuali di tipo racchiusa in parentesi angolari e separati da virgole e da una lista di valori, da assegnare agli argomenti, racchiusi tra parentesi tonde e separati da virgole.

```
var n : Int = max(5,6);
```

1.8.1 Overloading

Con overloading delle funzioni si intende la possibilità di fornire all'interno di un programma, eventualmente anche all'interno dello stesso package, multiple definizioni di funzioni aventi lo stesso nome, a patto che gli argomenti differiscano per quantità o per il tipo di almeno uno di essi. Due definzioni di funzioni aventi lo stesso nome si dicono una overload dell'altra. L'insieme di tutti gli overload di una certa funzione viene chiamato overload-set.

Analizziamo per esempio questi due overload per la funzione `max`, esse sono validi overload in quanto pur avendo lo stesso numero di argomenti, tali argomenti hanno tipo distinto, e dunque in sede di chiamata il compilatore analizzando i tipi degli argomenti concreti della chiamata sarà in grado, partendo da essi, di selezionare l'overload adatto.

```
func max(first : Int, second : Int) -> Int {
    if (first > second) {

        return first;
    }
    else {
        return second;
    }
}

func max(first : Float, second : Float) -> Float {
    if (first > second) {

        return first;
    }
    else {
        return second;
    }
}

func max(first : Number, second : Number) -> Number {
    if (first > second) {

        return first;
    }
    else {
        return second;
    }
}
```

Nel caso poi in cui esistano due overload entrambi validi, sarà scelto l'overload ritenuto più specifico, applicando queste politiche di selezione e filtraggio tra gli overload trovati.

- un overload non generico viene sempre preferito rispetto ad un overload generico, il numero di parametri di tipo non è rilevante
- un overload viene preferito ad un altro se i suoi parametri di tipo compaiono meno volte nei tipi dei suoi argomenti
- un overload viene preferito ad un altro se i suoi argomenti hanno tipi più complicati, ovvero hanno più parametri di tipo, oppure tali parametri di tipo sono a loro volta, ricorsivamente, più complicati
- un overload viene preferito ad un altro se tra i tipi dei suoi argomenti compaiono meno volte tipi definiti come union (anonime e non)
- un overload viene preferito ad un altro se il numero totale di casi coperti dalle union che compaiono tra i suoi argomenti è minore
- un overload viene preferito ad un altro se tra i tipi dei suoi argomenti e/o tra i parametri di tipo di questi ultimi vi sono meno conversioni di tipo

É possibile analizzare tutti gli aspetti appena elencati durante una fase apposita di preprocessing. Le funzioni quindi vengono valutate per la loro specificità prima che la ricerca dell'overload più appropriato abbia inizio. Tale ricerca, nota come overload-resolution è sostanzialmente un'operazione di scarto degli overload non compatibili con i tipi della chiamata, procedendo in ordine di specificità, il cui ordine è stato già stabilito a priori.

Qualora, alla fine della fase di scarto degli overload incompatibili, siano state trovate due definizioni ugualmente specifiche allora la chiamata viene definita ambigua e ciò porta ad un errore a tempo di compilazione.

In particolare, applicando la regola numero 3, si è in grado di affermare che tra i seguenti due overload di seguito riportati, vi è uno più specifico dell'altro ed esso è, come è lecito aspettarsi, l'overload che accetta un argomento di tipo `List<List<T>>`

IDENTIFICATIVO	***SPECIFICITÀ***
`func f<T>(x : List<List<T>>)`	#1
`func f<T>(x : List<T>)`	#2
`func f<T>(x : T)`	#3

Tabella 3: Comparazione specificità di overload per la funzione `f`

Di seguito è stata riportata una tabella dove sono state riportate alcuni overload della funzione `max`, raggruppate per specificità ed elencate dalla più alla meno specifica.

IDENTIFICATIVO	***SPECIFICITÀ***
`func max(first : Int, second : Int) -> Int`	#1
`func max(first : Float, second : Float) -> Float`	#1
`func max(first : Number, second : Int) -> Number`	#2
`func max(first : Int, second : Number) -> Number`	#2
`func max(first : Number, second : Float) -> Number`	#2
`func max(first : Float, second : Number) -> Number`	#2
`func max(first : Number, second : Number) -> Number`	#3
`func max<T>(first : Number, second : T) -> Number`	#4
`func max<T>(first : T, second : Number) -> Number`	#4
`func max<T>(first : T, second : T) -> T`	#5

Tabella 4: Comparazione specificità di overload per la funzione `max`

Durante la fase di overload-resolution per un'ipotetica chiamata alla funzione `max` con argomenti `Int` e `Float`, allora vi sarebbero più overload compatibili, ma nell'ambito dei soli overload compatibili, solo uno di essi è nettamente più specifico di tutti gli altri, ovvero l'overload corrisponente alla seguente firma:

```
func max(first : Number, second : Number) -> Number
```

1.8.2 Passaggio parametri

Il passaggio di parametri in Basalt è sempre per valore. Questo significa che, quando si passa un parametro ad una funzione, viene creata una copia del valore del parametro passato. Questo comportamento è in contrasto con il passaggio per riferimento, in cui la funzione riceve un riferimento alla variabile passata, permettendo alla funzione di modificare direttamente il valore della variabile passata.

Per poter accedere in scrittura ad una qualsiasi area di memoria esterna alla funzione, è necessario passare un puntatore (scalare o vettoriale) a tale area di memoria, così simulando un passaggio parametri per riferimento. Tale modo di operare è prassi consolidata in molti altri linguaggi di programmazione quali Go, C, Zig non solo.

```
func f(x : MyStruct, y : #MyStruct, z : ##MyStruct) {
    x.field = 10;
    y.field = 20;   // modifica osservabile dall'esterno
    z.field = 30;   // modifica osservabile dall'esterno

    var myStruct : MyStruct;

    x = myStruct;
    y = &myStruct;
    #z = &myStruct; // modifica osservabile dall'esterno
    z = &y;
}
```

Così come accade in C, modificare un puntatore all'interno di una funzione non comporta alcuna modifica all'indirizzo di memoria puntato. Ciò che si sta modificando è solo la copia locale dell'indirizzo. Modificare field di un puntatore invece, comporta una modifica effettiva all'indirizzo di memoria puntato, così come modificarne il valore dereferenziato. Analogamente a quanto appena visto per i puntatori scalari avviene per array e puntatori vettoriali, i quali vengono copiati.

```
func f(x : [10]MyStruct, y : #[10]MyStruct, z : $MyStruct) {
    x[0] = 10;
    (#y)[0] = 20;   // modifica osservabile dall'esterno
    z[0] = 30;      // modifica osservabile dall'esterno

    var array : [10]MyStruct;

    x = array;
    y = &array;
    #y = array;     // modifica osservabile dall'esterno
    z = &array;
}
```

1.8.3 Funzioni extern

Una funzione esterna, in generale, è una funzione che non è definita all'interno della compilation unit in cui viene invocata. Nel caso concreto del linguaggio Basalt, essa è dunque una funzione che si desidera invocare all'interno di un programma Basalt, ma che è definita in un altro linguaggio ad esempio in C o in C++.

Il linker, in fase di linking, si occuperà di risolvere il riferimento alla funzione esterna a patto che esso abbia a disposizione l'object file contenente la definizione della funzione. Tale object file può essere emesso dai compilatori C e C++.

Per poter invocare una funzione esterna, è necessario che essa sia dichiarata all'interno del programma Basalt utilizzando la keyword *extern* al posto della keyword *func*, e rimpiazzando il corpo della funzione con un simbolo di **=** seguito da una string literal detta nome-macchina, rappresentante il nome con cui tale funzione viene nominata nell'object file che si desidera linkare, seguita da un punto e virgola.

Una funzione extern non può essere generica, e non vi possono essere più funzioni esterne associate allo stesso nome macchina.

```
// C stdlib: double sqrt(double);
extern square_root(value : Float) -> Float = "sqrt";
```

Tale design consente di poter effettuare overloading di funzioni esterne, in quanto il nome da utilizzare per l'overloading è quello con cui essa è stata dichiarata, che è potenzialmente differente dal nome che essa ha nell'object file, e che è stato invece specificato in coda alla dichiarazione. Di seguito un esempio di quanto detto:

```
// C stdlib: int32_t putchar(int32_t);
extern print(char : Char) = "putchar";

func print(str : RawString) {
    var index : Int = 0;
    while (str[index] != '\0') {
        print(str[index]);
        index = index + 1;
    }
}

func print(str : String) {
    var index : Int = 0;
    while (index < len(str)) {
        print(str[index]);
        index = index + 1;
    }
}
```

1.9 Immutabilità

Con il termine immutabilità si intende la proprietà di un oggetto di non poter essere modificato una volta creato. In Basalt, le variabili sono mutabili di default, ovvero è possibile modificarne il valore in qualsiasi momento. Tuttavia, esistono molteplici costrutti in basalt che permettono di ottenere l'immutabilità. Ciò che è immutabile non può essere modificato nè tramite assegnazione, nè tramite assegnazione ai suoi field, se è una struct, alle sue celle, se è una slice, un array o una stringa, nè all'area di memoria da esso, se è un puntatore.

1.9.1 Valori Letterali

Un valore letterale, in inglese "literal", sono quei valori che sono scritti direttamente nel codice sorgente. Ad esempio `42`, `true`, `"hello"` sono tutti valori letterali. In Basalt, i valori letterali sono immutabili, il che dovrebbe essere ciò che l'utente si aspetta.

1.9.2 Espressioni elementari

Un'espressione elementare è un'espressione che ricade in una delle seguenti categorie:

- Applicazione di un operatore binario (come ad esempio l'operatore di somma `+` o l'operatore di confronto `==`)
- Applicazione di un operatore unario non legato alla manipolazione di puntatori (come ad esempio l'operatore di negazione logica `!`)

Espressioni di questo tipo sono sostanzialmente paragonabili ai valori letterali, nel senso che esse sono da immaginarsi sostituibili dal loro risultato senza alcuna perdita di significato pertanto esse sono da considerarsi immutabili.

1.9.3 Espressioni di sola lettura

Un'espressione di sola lettura è un'espressione che restituisce un valore temporaneo preso per copia. Questo significa che anche qualora essa fosse mutabile, la modifica non sarebbe osservabile in alcun modo. Un esempio di espressione di sola lettura è la chiamata ad una funzione che restituisce un puntatore. Il valore restituito dalla funzione è un valore temporaneo, preso per copia, esso si dice essere in sola lettura e non può essere modificato. Tuttavia non è corretto parlare di immutabilità in quanto ciò che si trova all'area di memoria indirizzata dal puntatore potrebbe essere modificato.

```
func get_ptr() -> #Int { return memory::alloc<Int>(); }
var number : Int = 7;

// Errore di compilazione -> modifica di espressione di sola lettura
get_ptr() = &number;

// Ok -> modifica del valore puntato (non immutabile)
#get_ptr() = number;
```

1.9.4 Costanti

Una costante è nient'altro che una variabile immutabile. A differenza di linguaggi come Java e Kotlin, la garanzia di immutabilità per le costanti si estende non solo a loro stesse ma anche ai loro membri se sono struct, agli oggetti da loro puntati se sono puntatori e così via. In linea generale è possibile affermare che una costante non può essere modificata in nessuna sua parte. La dichiarazione di una costante necessita di un'inizializzazione, e avviene in accordo alla seguente sintassi: `const <nome> : <tipo> = <valore>;`

```
const pi : Float = 3.14;
```

Ci si potrebbe chiedere quale sia l'utilità di un puntatore costante, dato che il puntatore stesso non può essere modificato così come il valore a cui punta. In effetti, l'utilità di un puntatore costante è limitata, specialmente se paragonata con quanto accade in C e C++, però la semantica della keyword `const` è molto differente. Un puntatore costante in Basalt può essere usato ad esempio come un alias, per tale scenario è riportato il seguente esempio:

```
struct Job {
    var name : String;
    var salary : Int;
}

struct Person {
    var name : String;
    var job : Job;
}

var person : Person = get_person();
    // Si assuma una funzione del genere esista

const job : #Job = &person.job;
```

In questo caso il puntatore `job` è costante, ciò implica che esso è immutabile e che non è possibile modificare nè il puntatore stesso nè ciò a cui punta utilizzando su di esso con operatore di dereferenziazione. Ciò non significa che ciò a cui il puntatore `job` punta sia destinato a non subire mai cambiamenti, in quanto accedendo in scrittura al campo `.job` della variabile `person`, la quale è mutabile, è possibile modificare il valore puntato da `job`.

In linguaggi come Java e Kotlin, le costanti (o `final` in Java) offrono garanzie di immutabilità solo per il riferimento, ma non per l'oggetto puntato. Questo significa che se si dichiara una costante di tipo `List`, è possibile modificare la lista aggiungendo o rimuovendo elementi, ma non è possibile assegnare un nuovo oggetto alla variabile costante. In Basalt, invece, una costante di tipo `List` non può essere modificata in nessun modo, nè aggiungendo nè rimuovendo elementi.

1.10 Assignments

Come in ogni linguaggio fortemente tipato, esistono regole ben precise che determinano quando è possibile assegnare un valore ad una variabile. In generale, è possibile assegnare espressioni non solo a variabili ma anche a vere e proprie espressioni.

Ci sono due tipi di vincoli che possono essere posti su un assegnamento: i vincoli di tipo ed i vincoli di immutabilità.

I vincoli di tipo sono il motivo stesso dell'esistenza di un linguaggio fortemente tipato, essi permettono di evitare errori di esecuzione dovuti ad un uso improprio delle variabili mediante un'analisi basata sul loro tipo dichiarato a tempo di compilazione.

I vincoli di immutabilità, invece, hanno a che fare con l'utilizzo delle costanti e/o delle espressioni immutabili (e.g. stringhe, numeri, ecc.).

In una assegnazione, in inglese "assignment", sono coinvolte due espressioni: l'espressione a sinistra dell'uguale (il target) e l'espressione a destra dell'uguale (il valore da assegnare). Ogni dichiarazione di costante o di variabile qualora inizializzata è implicitamente considerata un'assegnazione all'oggetto che si sta dichiarando.

1.10.1 Assignment semplici

Un assignment è ritenuto semplice se non coinvolge costanti o espressioni immutabili, e se il tipo del target non è nè un tipo Union, nè un tipo Array, nè un tipo Slice, nè un tipo Pointer, e se non ha parametri attuali di tipo.

Nelle condizioni appena descritte, l'assegnamento è possibile se e solo se i due nomi dei due tipi coinvolti corrispondono allo stesso identico tipo.

1.10.2 Assignment tra union

È utile immaginare una union come l'insieme dei tipi nominati direttamente o indirettamente al suo interno. È possibile assegnare ad una union espressioni il cui tipo è in tale insieme o espressioni il cui tipo è una union corrispondente ad un suo sottoinsieme.

Più formalmente, diciamo che è possibile assegnare ad espressioni di tipo union:

- espressioni del suo stesso tipo
- espressioni il cui tipo compare esplicitamente nel suo elenco dei tipi
- espressioni il cui tipo è assegnabile ad un tipo elencato nel suo elenco dei tipi
- espressioni il cui tipo è un'altra union, definita a partire da tipi a loro volta assegnabili

In altri termini, per le union e per le union soltanto si applica una politica di structural-compatibility, in luogo della name-equivalence.

1.10.3 Assignment tra puntatori

Quando il target di un'assignment è un puntatore scalare, è possibile assegnarvi solo espressioni che siano anch'essi puntatori scalari. Inoltre, è richiesto che i tipi degli oggetti puntati da entrambi coincidano strutturalmente, ovvero che essi siano identici oppure che essi siano union mutuamente assegnabili fra loro.

Ciò significa che anche se fosse possibile assegnare espressioni di tipo `V` a target di tipo `T`, qualora il viceversa non fosse vero, allora non sarebbe possibile assegnare espressioni di tipo `#V` a target di tipo `#T`.

Per quanto riguarda i puntatori vettoriali invece, vale quanto detto per i puntatori scalari ma con una dovuta precisazione, ovvero che è possibile assegnare ad un puntatore scalare `$T` un valore di tipo `[N]T`. Ciò è ovviamente vero in quanto in caso contrario verrebbe a mancare il senso stesso dei puntatori vettoriali, ovvero essere uno strumento per la gestione deglia array la cui dimensione è ignora a tempo di compilazione.

Per capire il motivo di tali restrizioni, è utile considerare il seguente esempio: si considerino due puntatori scalari `ptr : #Int` e `ptr2 : #(Int|Float)`, e si immagini cosa accadrebbe se fosse possibile assegnare `ptr` a `ptr2`. In tal caso, ci si ritroverebbe con un puntatore che punta ad un'area di memoria della dimensione sbagliata, il che potrebbe portare a comportamenti imprevedibili e a crash del programma, specialmente se si considera che tramite un riferimento a `ptr2` si potrebbe tentare di scrivere un valore di tipo `Float` in un'area di memoria che può contenere solo valori di tipo `Int`.

1.10.4 Assignment tra tipi generici

Considerando un qualunque tipo generico `Example`, qualora il target di un assignment fosse un tipo generico `Example<`T_1`, `T_2`, ...>`, vale la pena sottolineare che è possibile assegnare ad esso espressioni il cui tipo è `Example<`U_1`, `U_2`, ...>`, se e solo se i tipi T_i e U_i coincidono strutturalmente, ovvero se e solo se i tipi T_i e U_i sono mutuamente assegnabili fra loro.

1.10.5 Assignment Tra Array

Gli assignment a target di tipo array sono possibili solo se il tipo del valore da assegnare è anch'esso un array della medesima dimensione.

Essendo possibile vedere degli assignment tra array (e non slice), come una sequenza di assignment membro a membro, valgono le stesse regole che sono state discusse fin ora.

L'assegnazione di un array ad un altro array è un'operazione che potrebbe essere effettuata in tempo lineare qualora tali array contengano espressioni di tipo union e/o qualora l'architettura per cui si desidera compilare non supporti istruzioni SIMD (Single Instruction Multiple Data).

1.10.6 Assignment verso target immutabili

Assegnare un valore ad un target immutabile, ovvero ad un target costante o letterale (e.g. stringhe, numeri, ecc.), è proibito, e porta ad errori a tempo di compilazione.

É opportuno sottolineare che, in generale, un target è costante e dunque immutabile anche se esso è un membro di una costante di tipo struct, l'oggetto puntato da un puntatore costante, una cella di una slice costante o un elemento di un array costante

Il risultato di un'operazione binaria è sempre immutabile (ad esempio la somma di due numeri è immutabile) mentre il risultato di un'operazione unaria è immutabile solo se l'operando è immutabile in tutti i casi eccetto per l'operatore di dereferenziazione di un puntatore, il quale restituisce un valore mutabile per puntatori non costanti.

1.10.7 Assignment di espressioni immutabili

Assegnare espressioni immutabili a target immutabili è sempre permesso (ciò per costruzione può solo avvenire in sede di dichiarazione di una costante), assegnarli a target mutabili invece, è permesso solo se tale assegnamento non implica un legame del valore immutabile con un target mutabile.

Un legame di un valore immutabile con un target mutabile si ha ad esempio provando ad assegnare a tale target l'indirizzo di memoria del valore immutabile in formato di puntatore scalare o vettoriale.

Tale legame consentirebbe infatti di modificare il valore immutabile deferenziando il puntatore mutabile così ottenuto, il che è chiaramente in contrasto con la natura stessa del concetto di immutabilità.

1.10.8 Assignment verso espressioni di sola lettura

Un concetto affine a quello di immutabilità è quello di sola lettura. Un'espressione è considerata di sola lettura se essa è un valore mutabile ma il cui potenziale mutamento non porterebbe alcun effetto visibile all'esecuzione del programma in nessuna circostanza. Ad esempio, con il seguente frammento di codice si vuole mostrare una selezione di assignment, alcuni dei quali non validi in quanto tentano di assegnare un valore mutabile ad un target che in teoria sarebbe mutabile, ma che nel contesto di riferiemento è considerato immutabile in quanto il suo mutamento non porterebbe alcun effetto visibile all'esecuzione del programma.

```
get_pointer_to_int() = &x; // Errore
get_array_of_ints()[0] = 42; // Errore
get_slice_of_ints()[0] = 42; // Ok
get_pointer_to_struct().field = 42; // Ok
var x : Int = #get_pointer_to_int(); // Ok
```

1.11 Pseduo-polimorfismo

Con il termine polimorfismo in informatica, ci si riferisce alla capacità di un linguaggio di poter astrarre dal tipo concreto di un oggetto, permettendo di scrivere codice che possa essere applicato a tipi diversi, i quali condividono la stessa API.

Questo concetto viene spesso legato a doppio filo con quello di ereditarietà nei contesti di programmazione ad oggetti, in quanto l'API condivisa a tutti i tipi concreti su cui si desidera operare viene codificata nella forma della loro classe base (parent-class).

In Basalt, così come in Go, Rust e altri linguaggi moderni, è possibile ottenere gli stessi effetti del polimorfismo object-oriented senza dover ricorrere all'ereditarietà. In particolare Basalt utilizza un approccio unico nel panorama dei linguaggi di programmazione di basso livello, che sfrutta una feature chiamata *Common Features Adoption (CFA)* per implementare così una forma di pseudo-polimorfismo.

1.11.1 Common features adoption (CFA)

Con *Common Features Adoption*, abbreviato come CFA, ci si riferisce all'abilità del compilatore di generare un'overload di una funzione, basandosi sugli overload già esistenti, direttamente in sede di chiamata. In particolare, gli overload auto-generati implicitamente tramite CFA sono costruiti in modo tale da effettuare un dispatch a tempo di esecuzione sui tipi concreti degli argomenti di una chiamata a funzione, in modo da poter chiamare il corretto overload già esistente.

Si considerino ad esempio le seguenti definizioni di funzioni:

```
func half(x : Int) -> Int {
    return x / 2;
}

func half(x : Float) -> Float {
    return x * 0.5;
}
```

Se si effettuasse una chiamata alla funzione `half`, con un argomento di tipo `Int|Float`, il sistema non troverebbe un overload specifico. Esso però sarebbe in grado di generare un overload ad-hoc dietro le quinte simile a quanto riportato di seguito:

```
func half(x : Int | Float) -> Int | Float {
    if (x is Int) {
        return half(x as Int);
    } else {
        return half(x as Float);
    }
}
```

Tutto ciò che il programmatore dovrà fare sarà effettuare una chiamata a funzione. Qualora non esistesse un overload specifico per i tipi degli argomenti passati, il compilatore proverà a generare un overload definito per casi analizzando uno per uno tutti gli argomenti passati in chiamata da sinistra a destra.
Si considerino infatti i seguenti overload per la funzione `add`:

```
func add(x : Int, y : Int) -> Int {
    return x + y;
}

func add(x : Float, y : Int) -> Float {
    return x + utils::convert_to<Float>(y);
}

func add(x : Int, y : Float) -> Float {
    return utils::convert_to<Float>(x) + y;
}

func add(x : Float, y : Float) -> Float {
    return x + y;
}
```

In questo contesto, sarebbe possibile effettuare le seguenti chiamate a funzione:

CHIAMATA A FUNZIONE	***DEFINIZIONE CORRISPONDENTE***
`add(Int, Int)`	overload definito dall'utente
`add(Int, Float)`	overload definito dall'utente
`add(Float, Int)`	overload definito dall'utente
`add(Float, Float)`	overload definito dall'utente
`add(Int|Float, Int)`	overload generato tramite CFA
`add(Int|Float, Float)`	overload generato tramite CFA
`add(Int, Int|Float)`	overload generato tramite CFA
`add(Float, Int|Float)`	overload generato tramite CFA
`add(Int|Float, Int|Float)`	overload generato tramite CFA

Tabella 5: Comparazione specificità di overload per la funzione `add`

1.11.2 Implicazioni della CFA

Considerato quanto detto finora, è possibile trarre alcune conclusioni riguardo i benefici e le limitazioni che derivano dall'utilizzo della CFA in Basalt.

È necessario sottolineare come la CFA consenta di esporre al programmatore un'API basata sulle funzionalità comuni a tutti i tipi concreti che è possibile assegnare a un tipo base. Nel caso di Basalt, tale tipo base è una union. Questo significa che il programmatore può scrivere codice che opera su un tipo base, senza dover conoscere il tipo concreto, usando le funzionalità comuni a tutti i tipi concreti proprio come se tale tipo base fosse un'interfaccia di un linguaggio ad oggetti (e.g. Java, Kotlin, C#).

Ad esempio, sarà possibile chiamare la funzione `size` su una union dei tipi `List<T>`, `Tree<T>`, `HashSet<T>` come se tale union fosse un'interfaccia che esponesse tale funzionalità. Affinché ciò avvenga, sarà necessario che esista un overload definito dall'utente della funzione `size` per tutti i tipi citati.

1.11.3 Considerazioni e compromessi riguardanti la CFA

Risulta evidente come la CFA possa risultare peggiore dell'ereditarietà in termini di performance, in quanto ogni singola chiamata a funzione potrebbe comportare un controllo in tempo lineare sul numero di tipi concreti che è possibile assegnare ai vari tipi base degli argomenti.

Ciò nonostante, la CFA consente di scrivere codice estremamente flessibile e modulare, senza costringere il sistema a dover tener traccia di v-tables e/o object-headers di sorta, i quali sono invece necessari per implementare il polimorfismo tramite ereditarietà come ad esempio accade in Java, Kotlin, C#, i quali sono costi di overhead che impattano l'intera codebase, e non solo le sezioni che necessitano di usare il polimorfismo.

Basalt è ottimizzato per le situazioni dove non è necessario polimorfismo dinamico, in quanto ci si aspetta che solo una minima parte della codebase necessiti di tale feature, e pertanto, si ritiene che sia più giusto pagare un costo anche considerevole in termini di performance in situazioni mediamente rare pur di ottenere codice meglio performante in tutti gli altri scenari.

Si tenga poi a mente che per scenari dove il numero totale di tipi concreti da considerare per la generazione di overload CFA è molto contenuto (non più di 5 tipi), la CFA potrebbe risultare competitiva in termini di performance dato che il numero totale di istruzioni macchina corrispondenti a risolvere un riferimento a metodo in una v-table non è troppo dissimile dal numero di istruzioni necessarie per risolvere tale overload CFA.

2 Implementazione

2.1 Generalità sul processo di compilazione

Il processo di compilazione, per sua natura, è un processo sequenziale schematizzabile come una pipeline (catena di montaggio). Ogni fase della compilazione ha una responsabilità ben definita e produce un output che sarà l'input della fase successiva.

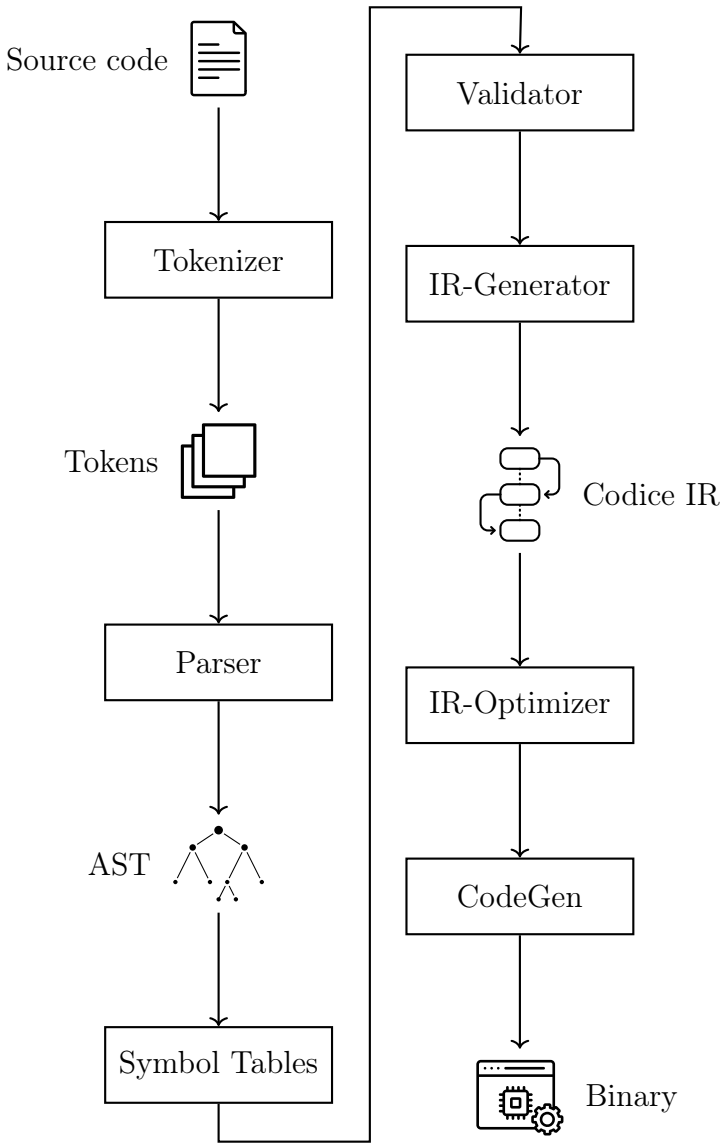


Figura 5: Pipeline del processo di compilazione

2.1.1 Tokenizzazione

Con tokenizzazione, si intende il processo di suddivisione del codice sorgente in *token*, ovvero in unità atomiche che rappresentano i componenti del linguaggio. Queste unità atomiche possono essere rappresentate come semplici stringhe, oppure come entità più ricche di informazioni, come ad esempio il nome del file sorgente dal quale sono stati estratti, la posizione all'interno del file (riga, colonna) e così via.

Il processo di tokenizzazione è il primo passo del processo di compilazione, ed è possibile implementare un tokenizzatore in diversi modi. In generale, è utile immaginare un tokenizzatore come un algoritmo iterativo che dato un input testuale continua a leggere caratteri finché essi non formano un token valido. Una volta che un token è stato riconosciuto, esso viene conservato in un'opportuna collezione.

Commenti, spazi e alcuni caratteri speciali sono generalmente ignorati dal tokenizzatore, nel senso che essi vengono correttamente riconosciuti ma non vengono conservati nella collezione.

Un token che non viene riconosciuto porta ad un errore a tempo di compilazione.

2.1.2 Parsing

Con *Abstract Syntax Tree* (AST) si intende una struttura dati ad albero che rappresenta un'espressione, uno statement o una definizione di una variabile in un linguaggio di programmazione. L'AST è in corrispondenza biunivoca con il codice sorgente, e viene utilizzato per rappresentare il codice sorgente in una forma più adatta per l'analisi e la manipolazione da parte del compilatore.

L'atto di trasformare una collezione ordinata di token in un AST **è chiamato *parsing***.

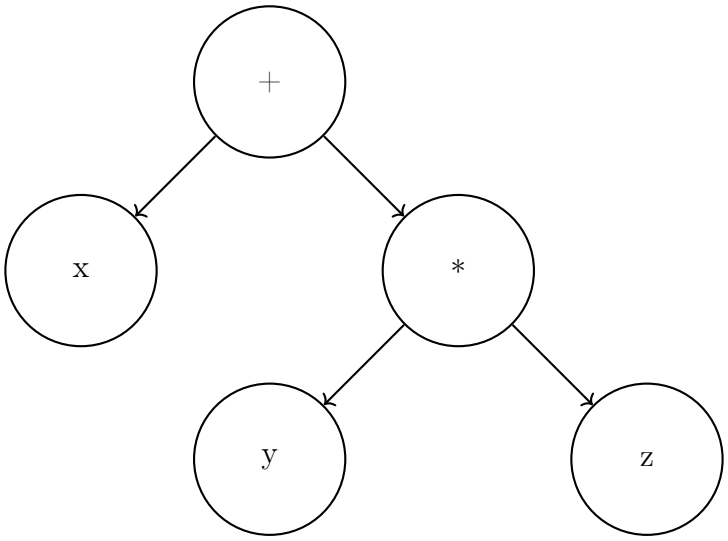


Figura 6: Esempio di generico AST per un'espressione aritmetica "x + y * z"

In pratica, il compilatore Basalt utilizza internamente AST simili al seguente. Tale albero è ben più strutturato in quanto ogni nodo rappresenta un'entità del linguaggio ben precisa.

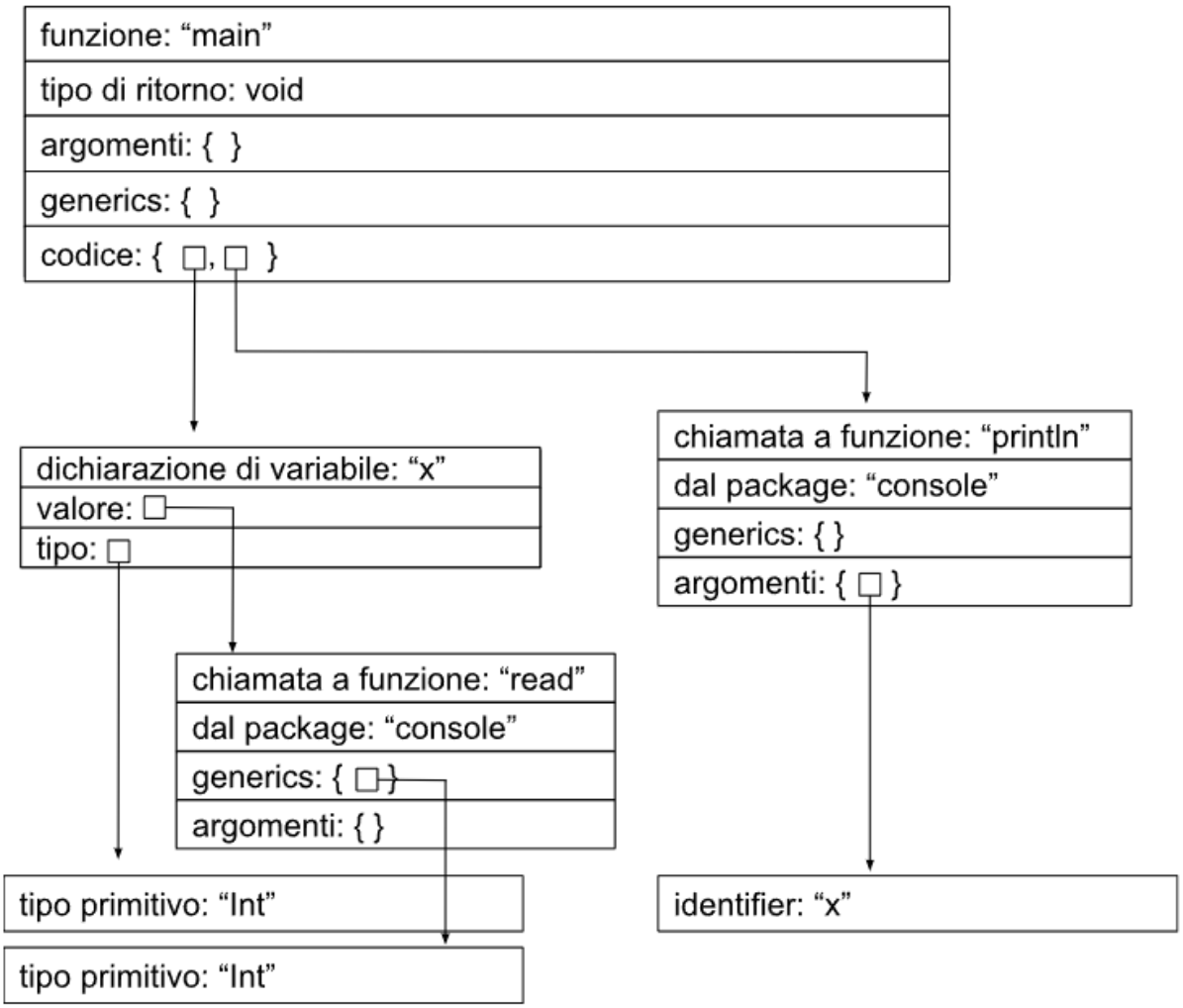


Figura 7: Esempio di AST per una funzione main che legge da riga di comando un numero intero e lo stampa subito dopo

Nei capitoli successivi sarà reso conto di come sia stato implementato il parsing in Basalt in dettaglio. Per completezza, si riporta che esistono due principali famiglie di algoritmi di parsing: i parser *LL* ed i parser *LR*. Tali parser differiscono per il modo in cui essi costruiscono l'AST. I parser LL costruiscono l'AST scansionando i token da sinistra a destra e costruendo il sottoalbero sinistro completamente prima di passare al token seguente (leftmost-derivation). I parser LR, invece, seppur scansionano i token da sinistra a destra, costruiscono l'AST modificando e rifinendo il sottoalbero destro man mano che scoprono nuovi token (rightmost-derivation).

2.1.3 Costruzione delle symbol-table

Le symbol-table (tabelle dei simboli), sono strutture dati che memorizzano le varie definizioni di funzioni e tipi presenti all'interno del programma in un formato che ne facilita il recupero.

Sostanzialmente si tratta di strutture dati chiave-valore in cui ad ogni chiave, che spesso è un AST, ad esempio relativo ad una chiamata a funzione o ad una type-signature, viene associata una definizione, anch'essa nella forma di AST, relativo alla definizione corrispondente.

Tipicamente esse sono costruite come wrapper su strutture dati più semplici, come ad esempio delle hash-map, ed implementano internamente una traduzione da AST (chiave per la symbol table) a stringa (chiave per la hash-map), con eventuali meccanismi di caching più o meno sofisticati per evitare di dover ricalcolare la chiave dell'hash-map ad ogni accesso.

2.1.4 Validazione ed analisi statica

La fase di validazione è una delle fasi più importanti del processo di compilazione. Essa consiste nel navigare con uno o più visitor l'AST generato dalla fase di parsing e verificare che esso sia corretto rispetto a determinate regole semantiche.

Basalt, effettua i seguenti controlli di validità sul codice sorgente:

- Aciclicità delle dipendenze dirette tra tipi: (Non esistenza di struct o union definite per ricorsione diretta)
- Non ambiguità dei tipi (Non esistenza di tipi con lo stesso nome e con lo stesso numero di parametri formali di tipo nello stesso package, o in package diversi ma importati in uno stesso file)
- Address sanitizing (Verifica che non si stia provando a calcolare l'indirizzo di un entità non allocata)
- Typechecking (Verifica che tutti i tipi usati esistano e siano coerenti con il contesto, controllo degli assignments per correttezza di tipo, risoluzione delle chiamate a funzione e controllo sui tipi dei parametri, sui generics e sul tipo di ritorno)
- Immutability-checking (Ispezione degli assignments e delle chiamate a funzioni ia fini di impedire modifiche a entità immutabili quali costanti e/o espressioni di sola lettura)
- Exit-path-checking (Ispezione dei flussi di esecuzione delle funzioni, ai fini di garantire l'assenza di codice irraggiungibile e la presenza di un return statement in tutti i possibili flussi di esecuzione per funzioni non void)

2.1.5 Conversione dell'AST in IR

Una volta che tutte le definizioni, in forma di AST, sono state validate, esse vengono convertite in IR (Intermediate Representation). Questa nuova forma di rappresentazione del codice sorgente permette di eseguire operazioni di ottimizzazione e di generazione del codice macchina in modo più efficiente.

Il codice macchina infatti è per sua natura lineare, ovvero è una sequenza ordinata di istruzioni. Il processore possiede un registro chiamato Program-Counter (PC) il quale tiene traccia dell'indirizzo dell'istruzione corrente, al termine di ogni istruzione il PC viene incrementato o decrementato in modo da puntare all'istruzione successiva.

La sfida della fase di traduzione dell'AST in IR consiste nel trasformare una struttura ad albero in una sequenziale, utilizzando le istruzioni di salto per rispecchiare fedelmente il flusso di esecuzione atteso.

Si consideri ad esempio il seguente frammento di codice, rappresentabile in forma di AST come mostrato in Figura 9 (si propone una rappresentazione semplificata):

```
if (cond1) {
    if (cond2) {
        console::println("cond1 && cond2");
    }
    else {
        console::println("cond1 && !cond2");
    }
}
else {
    console::println("!cond1");
}
```

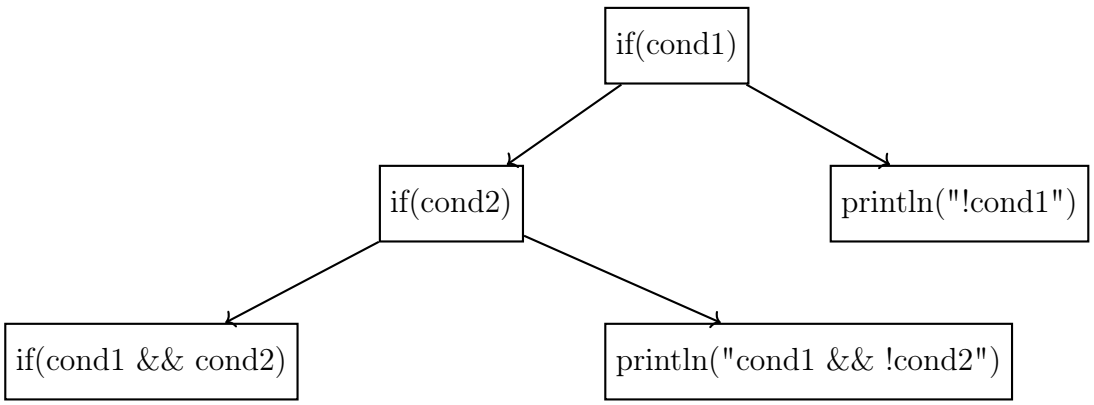


Figura 8: Esempio di AST **semplificato** per un doppio if-else annidato

L'AST nella figura precedente rappresenta un doppio if-else annidato. La conversione in IR di tale AST richiede l'utilizzo di istruzioni di salto condizionato per gestire correttamente il flusso di esecuzione. (anch'esso semplificato per motivi di leggibilità)

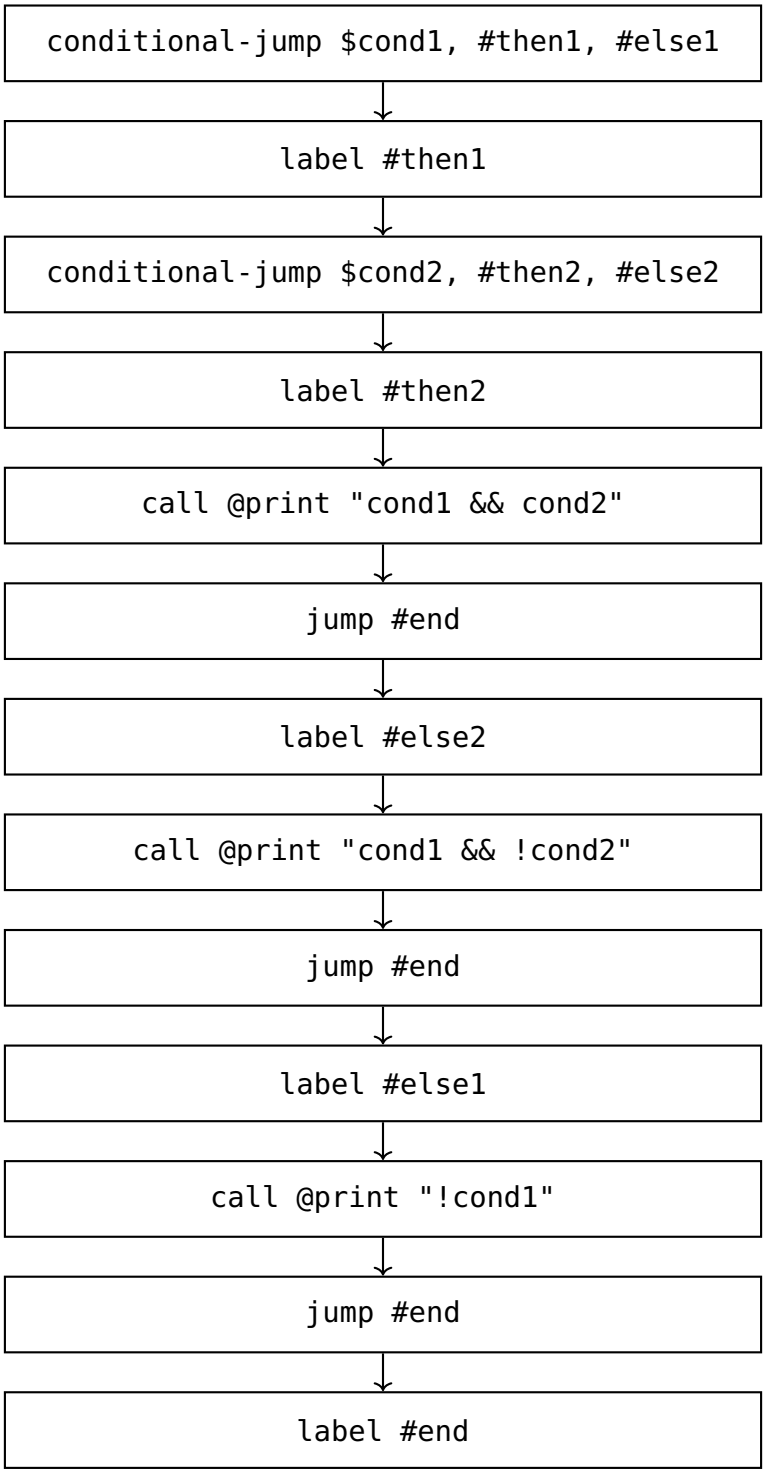


Figura 9: Esempio di IR **semplificato** per un doppio if-else annidato

2.1.6 Ottimizzazione dell'IR

Una volta che l'IR è stato generato, esso si assume semanticamente corretto, e si può finalmente perdere traccia dell'AST, delle symbol-table e di tutte le altre strutture dati intermedie, comprese tutte le informazioni relative alle coordinate dei vari elementi del programma all'interno del sorgente.

La fase di ottimizzazione dell'IR consiste nel migliorare il codice IR mediante l'applicazione di trasformazioni che ne riducano la complessità e ne migliorino le prestazioni ma che ne lasciano invariati gli effetti osservabili.

Le ottimizzazioni possono essere di vario tipo, alcune delle più comuni sono:

- **Constant folding**: Calcolo di espressioni costanti in fase di compilazione.
- **Common subexpression elimination**: Rimozione di espressioni comuni
- **Loop minimization**: Estrazione di codice dal corpo dei cicli iterativi
- **Inlining**: Sostituzione di chiamate a funzione con il corpo della funzione stessa
- **Register optimization**: Utilizzo più efficiente dei registri del processore
- **Code Reordering**: Riordinamento di istruzioni non dipendenti fra loro per poter raggruppare istruzioni che possono essere eseguite in parallelo in un'unica sezione
- **SIMD**: Utilizzo di istruzioni SIMD (Single Instruction Multiple Data) al posto di sequenze di istruzioni normali ripetute (Possibile solo per alcune architetture)

Una trattazione dettagliata dell'argomento è fuori dallo scopo di questo documento, in quanto è un ambito estremamente ampio ed in continua espansione.

2.1.7 Conversione dell'IR in codice macchina

Il codice IR, una volta ottimizzato, viene convertito in codice macchina mediante un processo chiamato *code-generation*. Il codice IR per sua natura è estremamente simile al codice macchina in struttura e semantica, e quindi la conversione è un processo relativamente semplice.

Ogni istruzione IR è direttamente rimpiazzabile con una o più istruzioni macchina che eseguono la stessa operazione. A differenza del codice IR, che è cross-platform, il codice macchina è specifico per l'architettura del processore su cui si intende eseguire il programma.

Le istruzioni macchina sono codificate in linguaggio binario, e sono scritte su file in modo incrementale man mano che vengono generate. Il file risultante è un file oggetto che è possibile linkare ed eseguire.

2.2 Frontend/backend compiler-frameworks

Adottando una terminologia diffusasi primariamente nel contesto della programmazione web, è possibile introdurre i concetti di frontend e backend in un compilatore. Con il frontend di un compilatore si intende l'insieme di componenti software (package, classi, metodi, funzioni) che si occupano di implementare le fasi di tokenizzazione e di parsing del codice sorgente. Con il termine backend, invece, si fa riferimento all'insieme di componenti software che si occupano di implementare le fasi di ottimizzazione dell'IR e di codegen.

Con il passare del tempo, ci si è accorti che estrapolare dalle codebase dei vari compilatori i loro frontend e backend, per poi generalizzarli e renderli riutilizzabili, avrebbe portato a una maggiore efficienza nello sviluppo di nuovi linguaggi. Questo ha portato alla nascita di frameworks per la creazione di compilatori, che offrono funzionalità più o meno avanzate negli ambiti del frontend e del backend, in modo da lasciare liberi i designer di un dato compilatore di concentrarsi sull'analisi statica.

Questo ha consentito, ad esempio, a linguaggi come Rust di svilupparsi in termini di features semantiche, quali ad esempio il borrow checker, senza dover preoccuparsi prima di implementare un backend perfettamente funzionante da zero.

2.2.1 LLVM: Low Level Virtual Machine

Ciò che accomuna Basalt, Rust, Zig e molti altri linguaggi moderni è l'adozione di LLVM come backend-framework. LLVM è un progetto open-source che si occupa di offrire una API per costruire manualmente rappresentazioni IR per le varie istruzioni di un programma, e di fornire un set di tool per ottimizzare e generare codice macchina a partire da tali rappresentazioni IR.

In altri termini, LLVM si occupa di implementare le fasi di IR-optimization e codegen discusse in precedenza, essendo cross-platform, ovvero supportando varie CPU e varie architetture (x86, ARM, Web-Assembly, etc...).

il team di LLVM supporta ufficialmente C e C++, ma vari porting non ufficiali esistono per molteplici altri linguaggi.

LLVM ha infine reso possibile la creazione di cling, un interprete C/C++ basato sul compilatore clang, il quale utilizza LLVM per effettuare JIT-compilation (compilazione "just in time", ovvero al volo) di codice C/C++. Tale strumento è tutt'ora di fondamentale importanza per la comunità scientifica, ed è in utilizzo presso il CERN.

2.2.2 Introduzione ad LL (LLVM-IR)

LLVM, come già detto nel paragrafo precedente, è un progetto composto da un insieme di librerie, e da alcuni tool. Due di questi tool sono `lli` ed `llc`, rispettivamente un interprete IR e un compilatore IR.

Questi due strumenti possono essere usati per eseguire codice IR, fornito sotto forma di file di testo con estensione `.ll`. Analizzare la sintassi e, più in generale, la semantica di tale formato testuale renderà più facile comprendere l'utilizzo della API nei capitoli successivi.

Un simbolo globale deve essere preceduto dal carattere speciale "@", ad esempio la funzione che fa da entry-point per l'esecuzione del programma è definita come `@main`.

Per definire una funzione è necessario specificare il tipo di ritorno, il nome della funzione, e la lista dei parametri. Il corpo della funzione è racchiuso tra parentesi graffe.

```
define i32 @main() {
  entry:
    ret i32 0
}
```

Una funzione in LL è fatta da uno o più blocchi, ognuno dei quali è composto da una lista di istruzioni. Ogni blocco termina con un'istruzione di return o di salto. Nella funzione `@main` definita sopra, `entry` è una label, e definisce l'ingresso nell'omonimo blocco. Il blocco entry è un blocco speciale in quanto esso è il primo ad essere eseguito in ogni funzione.

L'istruzione `ret` effettua il return e, pertanto, sancisce la fine del blocco corrente. Il valore restituito deve essere dello stesso tipo specificato nella firma della funzione e tale tipo va esplicitamente specificato.

I nomi dei simboli possono contenere spazi e/o caratteri speciali purchè racchiusi tra doppi apici. Per effettuare un return da una funzione void servirà scrivere `ret void`.

```
define void %"example void function"() {
  entry:
    ret void
}
```

L'utilizzo di "%" in luogo di "@" identifica un simbolo come locale invece che globale, pertanto esso non sarà visibile all'esterno del file `.ll` in cui esso è definito.

2.2.3 Implementazione di strutture dati in LL

È possibile definire delle struct in LL, le quali sono simboli come le funzioni e possono essere globali o locali. Di seguito è riportato un esempio di definizione di due struct locali, che rappresentano rispettivamente uno stack (struttura dati) ed un suo nodo.

```
%Stack = type {
    %Node*,
    i64
}

%Node = type {
    %Node*,
    i32
}
```

Volendo mostrare solo un esempio di come è possibile usare le struct definite sopra, si riporta la definizione di una funzione che stampa tutti gli elementi di uno stack.

Nel seguente frammento di codice `.ll` si può notare che, per chiamare una funzione, si utilizza l'istruzione `call`, la quale richiede l'esplicita specificazione del tipo di ritorno della funzione chiamata. L'istruzione `br` è una istruzione di salto.

```
define void @printStack(%Stack* %stack) {
  entry:
    %cursorPtr = call %Node* @getHead(%Stack* %stack)
    br label %loopCondition

  loopCondition:
    %cursor = load %Node*, %Node** %cursorPtr
    %condition = icmp ne %Node* %cursor, null
    br i1 %condition, label %loopBody, label %loopExit

  loopBody:
    %currentNumberPtr = call i32* @getNumber(%Node* %cursor)
    %currentNumber = load i32, i32* %currentNumberPtr
    call void @printInt(i32 %currentNumber)
    %currentNextPtr = call %Node* @getNext(%Node* %cursor)
    %currentNext = load %Node*, %Node** %currentNextPtr
    store %Node* %currentNext, %Node** %cursorPtr
    br label %loopCondition

  loopExit:
    ret void
}
```

2.2.4 Utilizzo della memoria in LL

In LL, la memoria è gestita in modo drasticamente diverso rispetto a come avviene nei linguaggi di programmazione di uso comune.

Nei linguaggi di programmazione tipici una variabile è associata ad un indirizzo di memoria ed è possibile assegnare valori a tale variabile riferendosi ad essa con il suo nome. Quando servirà leggere il valore di tale variabile, sarà sufficiente, ancora una volta, riferirsi ad essa con il suo nome.

In LL ciò non avviene, in quanto le variabili non hanno un vero e proprio indirizzo di memoria e sono invece solo dei simboli. Si consideri ad esempio il seguente frammento di codice Basalt e la sua fedele traduzione in LL:

```
var number : Int = 42;
console::println(number);
```

```
example_block:
  %numberAddress = alloca i64
  store i64 42, i64* %numberAddress
  %numberValue = load i64, i64* %numberAddress
  call void @"console::println<Int>"(i64 %numberValue)
```

Si noti come la variabile `number` sia stata tradotta come la `numberAddress` e `numberValue`. La variabile `numberAddress` è un puntatore ad una locazione di memoria e la variabile `numberValue` è il valore numerico contenuto in tale area di memoria.

Nè `numberValue` nè `numberAddress` in questo caso hanno dei veri e propri indirizzi di memoria ad essi associati. Essi sono solo dei simboli a cui sono associati dei valori.

La gestione della memoria di LL può sembrare controintuitiva al programmatore medio, ed è proprio per questo che anche se LL è un vero e proprio linguaggio, esso non è considerato veramente utilizzabile. Si è infatti abituati a ragionare in termini di variabili, intese come coppie indirizzo/valore, ed un approccio simile è facilmente classificabile come innaturale.

È bene ricordarsi però che i calcolatori utilizzano internamente una rappresentazione analoga a quella proposta da LL ed è proprio per questo che LL è facilmente ed efficientemente traducibile in linguaggio macchina nativo per molteplici architetture.

2.2.5 ANTLR: Another Tool for Language Recognition

ANTLR è un progetto open-source che si occupa di offrire uno strumento per generare codice in vari linguaggi, a partire da grammatiche e vocabolari in formato g4, che rappresentano la sintassi di un linguaggio di programmazione. Il codice generato da ANTLR è difatti un'intera unità di frontend provvista di tokenizer (lexer), parser e anche di un visitor, ovvero una classe astratta che può essere estesa per implementare agevolmente meccanismi di navigazione dell'AST.

Assieme al codice autogenerato, è necessario aggiungere al progetto una dipendenza, ovvero il runtime di ANTLR, che si occupa sostanzialmente di fornire le funzionalità di base sulle quali il codice autogenerato si basa.

2.2.6 ANTLR: Vocabolari e grammatiche

ANTLR lavora con due file di testo in formato `.g4`, che rappresentano rispettivamente il vocabolario e la grammatica del linguaggio.

Un vocabolario è un file contentente le regole di tokenizzazione in forma di espressioni regolari. Ogni regola è composta da un nome, seguito dai due punti, e da un'espressione regolare oppure un exact-match delimitato da apici.

Il seguente frammento di vocabolario `.g4` è un'estratto dal vocabolario di Basalt, dove si definiscono i token `ID` e `TYPENAME`, rispettivamente per gli identificatori (nomi di variabili, costanti, funzioni) e per i nomi dei tipi.

```
ID       : [a-z][a-zA-Z_0-9]* ;
TYPENAME : [A-Z][a-zA-Z_0-9]* ;
```

Una grammatica è un file contentente le regole di parsing del linguaggio. Ogni regola è composta da un nome, seguito dai due punti, e da una sequenza di token e/o regole eventualmente combinate.

Il seguente frammento di grammatica `.g4` è un'estratto dalla grammatica di Basalt, dove si definiscono le regole per il parsing di variabili e costanti.

```
variableDeclaration
    : VAR ID COLON typeSignature ASSIGN expr SEMICOLON
    | VAR ID COLON typeSignature SEMICOLON
    ;

constDeclaration
    : CONST ID COLON typeSignature ASSIGN expr SEMICOLON
    ;
```

2.2.7 ANTLR: Generazione del frontend

Una volta spostatosi nella directory dove sono presenti i file `.g4`, è possibile generare il frontend da riga di comando utilizzando l'apposito cli-tool di ANTLR, ovvero `antlr4`.

Per generare il frontend, è necessario specificare il linguaggio di destinazione, il quale può essere Java, Python, C++, ecc...

```
antlr4 -Dlanguage=Cpp -no-listener -visitor *.g4
```

2.2.8 Considerazioni generali

L'utilizzo di frameworks nello sviluppo di compilatori è una pratica ormai consolidata, anche se vale la pena chiedersi se tale pratica sia sempre conveniente.

LLVM è un framework largamente usato da numerosi compilatori per linguaggi di successo, diffusi, performanti, sicuri ed efficienti. LLVM ha dimostrato di essere un framework affidabile e robusto, e soprattutto la sua larga user-base ne garantisce la manutenzione e l'aggiornamento costante, il quale a sua volta garantisce la compatibilità con le ultime versioni di CPU e architetture, che in un periodo come questo è molto importante data la proliferazione di nuove archietture ARM / RISC-V.

L'egemonia dell'architettura x86 sarà, con ogni probabilità, messa in discussione nei prossimi decenni, pertanto lo sviluppo di un backend da zero, richiederebbe uno studio approfondito di molti standard, diversi fra loro, ed in continuo aggiornamento.

In definitiva, l'utilizzo di LLVM come backend-framework è una scelta proficua, che permette di concentrarsi su ciò che realmente conta, ovvero le symbol-tables e l'analisi statica.

Per quanto ANTLR invece, la situazione è un po' diversa. ANTLR è un framework molto potente, ma vale la pena sottolineare che esso, occupandosi di frontend a tutto tondo, si occupa anche della gestione dei commenti e nell'emissione degli errori di sintassi. Questo comporta un minor controllo sul tipo di feedback da dare all'utente, il quale potrebbe ricevere messaggi di errore non chiari o addirittura fuorvianti.

Inoltre, utilizzare ANTLR in C++, porta a dei problemi di compatibilità per via dell'utilizzo di alcune features deprecate dei vecchi standard C++ all'interno del codice autogenerato. Ciò è facilmente risolvibile mediante correzione manuale, ma ciò andrebbe a inficiare la facilità di manutenzione e di aggiornamento, rendendo difficile modificare la grammatica del linguaggio (ogni modifica della grammatica richiederebbe una nuova generazione e dunque una nuova correzione manuale).

2.3 Sviluppo del compilatore Basalt

Alla luce di quanto detto nei capitoli precedenti, si è deciso di sviluppare il compilatore per il linguaggio di programmazione Basalt in C++. Questa scelta è stata fatta per diversi motivi, tra cui:

- **Facilità di utilizzo del compilatore**: É stato dato un particolare peso alla facilità di utilizzo del prodotto finito. Un compilatore scritto in Java ad esempio, avrebbe richiesto l'installazione di una JVM, rendendo il prodotto meno accessibile (Considerazioni analoghe possono essere fatte per C# o simili). Inoltre, in C++ è possibile creare eseguibili linkati staticamente, semplificando il processo di distribuzione del compilatore
- **Performance**: C++ è un linguaggio altamente performante, requisito fondamentale per un compilatore. Un compilatore scritto in un linguaggio più lento avrebbe richiesto tempi di compilazione più lunghi, rendendo l'esperienza dell'utente finale meno piacevole
- **Supporto di LLVM**: L'adozione di LLVM è stata una scelta compiuta fin dalle primissime fasi di design del compilatore. La pool di linguaggi supportati da LLVM è comunque piuttosto ristretta, rendendo C++ una scelta quasi obbligata. Inoltre, LLVM è scritto in C++, rendendo la scelta ancora più naturale.

2.3.1 Doppia repository: Con e senza ANTLR

Il progetto Basalt è stato suddiviso in due repository distinte. Una di esse, denominata *Basalt*, contiene il compilatore vero e proprio, scritto in C++ e basato su LLVM. L'altra, denominata *unina-Basalt*, contiene l'adattamento della codebase principale ad ANTLR4.

Sono state create due repository per toccare con mano i vantaggi e gli svantaggi dell'utilizzo di ANTLR, i quali sono stati già discussi in precedenza. É opportuno evidenziare come se ANTLR non causasse problemi di compatibiltà che inficiano la riproducibiltà delle build, la repository basata su ANTLR avrebbe rimpiazzato la repository principale.

Repository	***URL***
`unina-Basalt`: basata su ANTLR4	`www.github.com/fDero/unina-Basalt`
`Basalt`: principale	`www.github.com/fDero/Basalt`

Tabella 6: Repository github

2.3.2 Build automatizzata

Posto che la build del compilatore nella sua versione basata su ANTLR è manuale, e può essere condotta solo usando specificamente il compilatore gcc con lo standard 17, la build del compilatore Basalt nella sua versione proposta sulla repository principale è automatizzata.

La build automatizzata utilizza conan, un package manager per C++, per scaricare le dipendenze del progetto (LLVM e libxml2), le quali sono poi compilate in locale automaticamente solo al primo utilizzo.

Scaricate le dipendenze, il progetto viene compilato con cmake, che è uno strumento che consente di effettuare build incrementali di progetti C e C++.

2.3.3 Installer per Windows x86

Per facilitare l'installazione del compilatore Basalt su sistemi Windows x86, è stato creato un installer automatico nel formato msi (Microsoft Installer).

Tale installer posiziona l'eseguibile (staticamente linkato) del compilatore Basalt nella directory `%Program Files%\basalt\<version>`, e aggiunge la directory alla variabile d'ambiente `PATH`. In fase di disinstallazione, che avviene dal pannello di controllo, rimuove l'eseguibile e la directory dalla variabile d'ambiente.

Tale installer è stato creato con WiX Toolset, uno strumento del `.NET` framwork di Microsoft. Esso mostra in fase di installazione una End-User License Agreement (EULA) in formato rtf (Rich text format).

2.3.4 Package per linux

Per facilitare l'installazione del compilatore Basalt su sistemi Linux, è stata predisposta la pacchettizzazione per snapcraft (package manager di canonical).

Snapcraft è uno strumento che consente di creare pacchetti snap, che sono pacchetti software sottoforma di container, che contengono tutte le dipendenze necessarie per l'esecuzione del software. Snapcraft è nato nell'ecosistema Ubuntu, ma è possibile utilizzarlo anche in altre distribuzioni Linux basate su Debian, e il supporto è attualmente in espansione anche per altre distribuzioni.

Il pacchetto non è attualmente sulle repository ufficiali di snapcraft, ma è possibile installarlo scaricandolo manualmente dalla repository github del progetto. Il nome per la futura pubblicazione di Basalt sui repository ufficiali di Snapcraft è già stato riservato.

2.4 Soluzioni implememtative per C++

C++ è un linguaggio di programmazione molto complesso, ma anche molto versatile. Esso ben si presta ad implementare design pattern peculiari che in altri linguaggi non sarebbero possibili.

2.4.1 Variant

Il design pattern *Variant* è un pattern che permette di implementare una union di tipi anche quando alcuni di essi sono sprovvisti di un costruttore di default (senza argomenti). In C++17 è stato introdotto il tipo `std::variant`, che permette di fare esattamente questo. All'interno della codebase di Basalt è possibile trovare una classe `SmartVariant` che è un wrapper su `std::variant` e che espone una API più consona all'utilizzo specifico all'interno del progetto.

2.4.2 Type-Erasure

Il design pattern *Type-Erasure* è un pattern che permette di avere vero e proprio polimorfismo object-oriented ma esponendo un API senza puntatori ed eventualmente ottimizzata con *copy on write*.

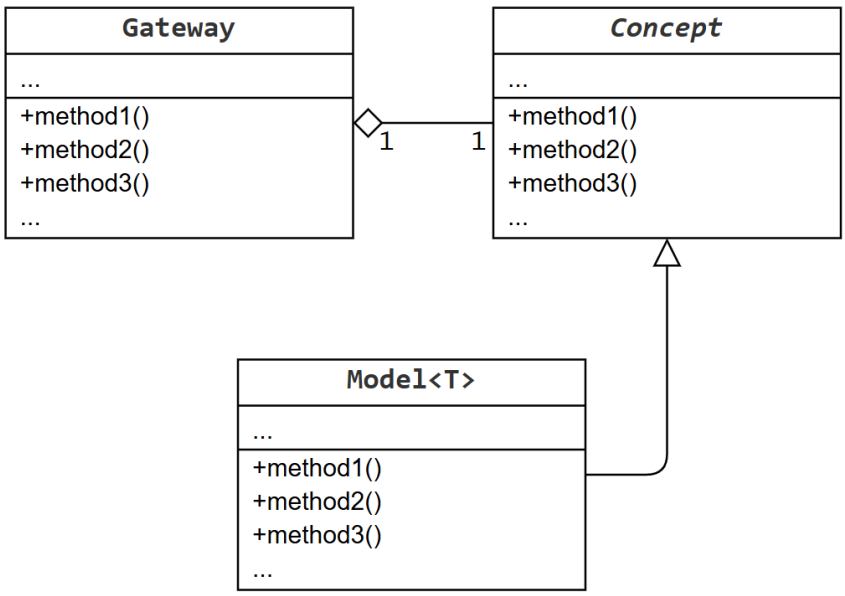


Figura 10: UML class diagram del pattern Type-Erasure

La classe `Gateway` è un esempio di implementazione di Type-Erasure. Essa viene utilizzata dai client come supertipo di tutti gli oggetti che implementano dei metodi pubblici chiamati `method1()`, `method2()`,`method3()`, `method4()`, etc.

Internamente essa possiede un puntatore a `Concept`, a cui è possibile assegnare puntatori ad oggetti di tipo `Model<T>` per ogni scelta di `T`.

A patto che `T` esponga la corretta API, la classe `Model<T>` può implementare i metodi chiamandoli direttamente su di un'istanza di `T` che essa possiede come membro interno.

Assegnare un oggetto di tipo `T` ad un `Gateway` è possibile grazie alla ridefinizione dell'operatore `=` nella classe `Gateway`. Assegnando un oggetto di tipo `T` a un oggetto di tipo `Gateway` allora il suo puntatore a `Concept` sarà aggiornato per puntare all'indirizzo di memoria di una cella appositamente allocata per contenere un `Model<T>` costruito a partire dall'oggetto `T` ricevuto.

```
#define ABSTRACT 0
template <typename T>
class Concept {
    public:
        virtual void method1() = ABSTRACT;
        virtual void method2() = ABSTRACT;
        //...
};

template <typename T>
class Model : public Concept {
    private:
        T object;
    public:
        Model(T obj) : object(obj) {}
        void method1() override { object.method1(); }
        void method2() override { object.method2(); }
        //...
};

class Gateway {
    private:
        std::unique_ptr<Concept> concept = nullptr;

    public:
        template<typename T>
        Gateway& operator=(T obj) {
            concept = std::make_unique<Model<T>>(obj);
        }

        void method1() { concept->method1(); }
        void method2() { concept->method2(); }
        //...
}
```

questo pattern consente di assegnare a oggetti di tipo `Gateway` oggetti di tipo `T` che implementano l'API corretta, e di chiamare i metodi di `T` attraverso l'oggetto di tipo `Gateway` senza usare esplicitamente puntatori, dereferenziazioni e senza dover deallocare manualmente nulla.

2.4.3 Polymorph

Il design pattern *Polymorph* è un pattern creato specificamente per le esigenze della codebase di Basalt. Esso è derivante dal pattern Type-Erasure, ma con alcune sostanziali differenze.

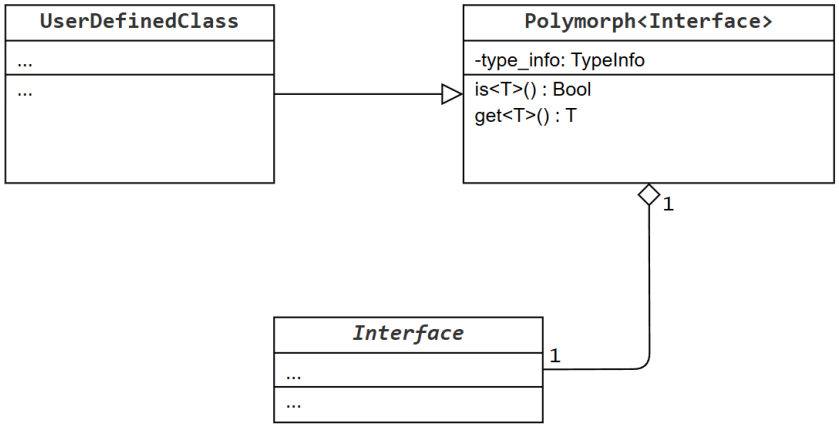


Figura 11: UML class diagram del pattern Polymporph

Ogni classe definita dal client (ad esempio `UserDefinedClass`) deve estendere la classe `Polymorph<Interface>`. Facendo così, essa erediterà i metodi concreti `is` e `get` che permettono di fare *downcasting* dell'oggetto a runtime.

Contemporaneamente, sarà possibile assegnare a oggetti di tipo `UserDefinedClass` oggetti di tipo `T` purchè `T` estenda la classe `Interface` mediante ereditarietà.

Assegnando un oggetto di tipo `T` a un oggetto di tipo `UserDefinedClass` allora, essa utilizzerà l'operatore di assegnazione definito in `Polymorph` per assegnare ad un puntatore ad `Interface` l'oggetto che si desidera assegnare ed aggiornare delle variabili interne per mantenere traccia del tipo dell'oggetto assegnato.

All'occorrenza si potrà interrogare la classe `UserDefinedClass` per sapere se contiene un oggetto di tipo `T` o meno, utilizzando il metodo `is` e fare il *downcasting* dell'oggetto a runtime utilizzando il metodo `get`.

Di seguito è riportato un esempio di implementazione del pattern *Polymorph*, così come è stato appena descritto, al netto della classe `UserDefinedClass`:

```
class Interface {
    public:
        //...
};

template <typename Interface>
class Polymorph {
    private:
        std::shared_ptr<Concept> ptr = nullptr;
        TypeInfo type_info;

    public:
        template <typename Implementation>
        Polymorph& operator=(Implementation obj) {
            ptr = std::make_shared<Interface>(obj);
            type_info = TypeInfo::from<Implementation>();
        }

        template <typename Implementation>
        bool is<>()
            requires(std::is_base_of_v<Interface, Implementation>)
        {
            return TypeInfo::from<Implementation>()
                == type_info;
        }

        template <typename Implementation>
        Implementation& get()
            requires(std::is_base_of_v<Interface, Implementation>)
        {
            if (is<Implementation>()) {
                return *static_cast<Implementation*>(ptr.get());
            }
            throw std::runtime_error("Invalid downcast");
        }
}
```

`Interface` in questo caso serve solo ad essere estesa da tutte e sole le classi che si desidera assegnare al `Polymorph`. L'estendere `Interface` è necessario in quanto è stato introdotto tale vincolo usando `requires(std::is_base_of_v<T,U>)` (C++20 concepts).

2.4.4 Notazioni e diagrammistica

Dato che questi design pattern servono a rendere il codice più leggibile e mantenibile, astraendo rispetto a come un certo comportamento polimorfico sia stato implementato, non renderebbe giustiza alla codebase di Basalt l'essere rappresentata con un UML class diagram tradizionale, in quanto essa risulterebbe artificiosamente più complessa. Per questo motivo, si è deciso di utilizzare la notazione UML di sotto-tipo (is-a relationship) per rappresentare non solo l'ereditarietà, ma anche l'implementazione di questi design pattern i quali rispecchiano proprio una relazione di sotto-tipo.

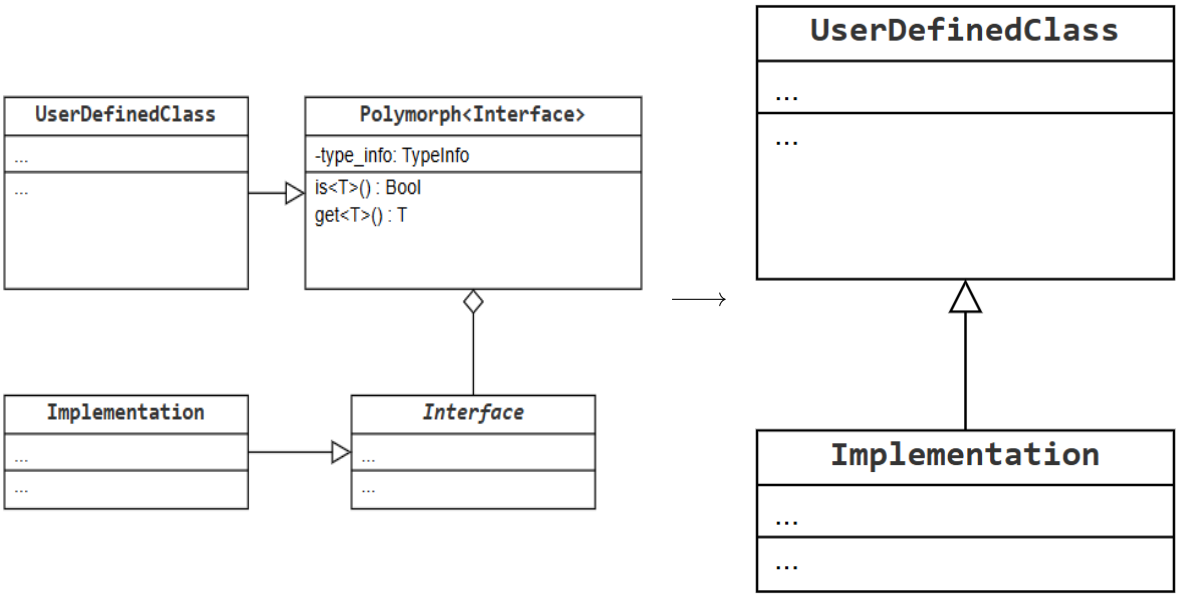


Figura 12: Traduzione UML del pattern Polymorph

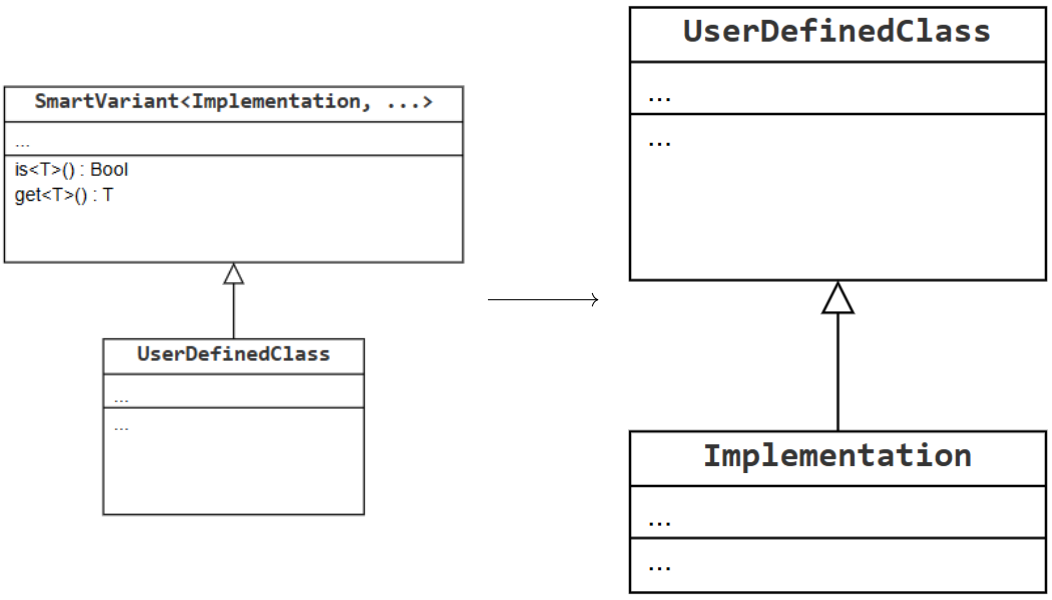


Figura 13: Traduzione UML del pattern Variant

2.5 Frontend: Tokenizzazione, Parsing, AST

In questo capitolo saranno approfondite le scelte implementative intraprese durante la realizzazione del frontend del compilatore.

2.5.1 Utilizzo di ANTLR nella repository *unina-Basalt*

Come già detto in precedenza, per la repository *unina-Basalt* è stato utilizzato ANTLR4 come frontend framework. Tale scelta implica l'utilizzo di un parser autogenerato.

In particolare, lo strumento da riga di comando `antlr4` è stato utilizzato per generare il codice sorgente della classe `BasaltParserVisitor`, il quale estende `AbstractParseTreeVisitor`.

La classe `BasaltParserVisitor` è stata pensata dagli sviluppatori di ANTLR per essere estesa da una implementazione concreta. Tale classe infatti espone dei metodi astratti che devono essere implementati per poter effettuare il parsing del codice sorgente.

Nel caso specifico di Basalt, la classe `BasaltParserVisitor` è stata estesa dalla classe `ConcreteBasaltParserVisitor`, la quale implementa i suddetti metodi astratti.

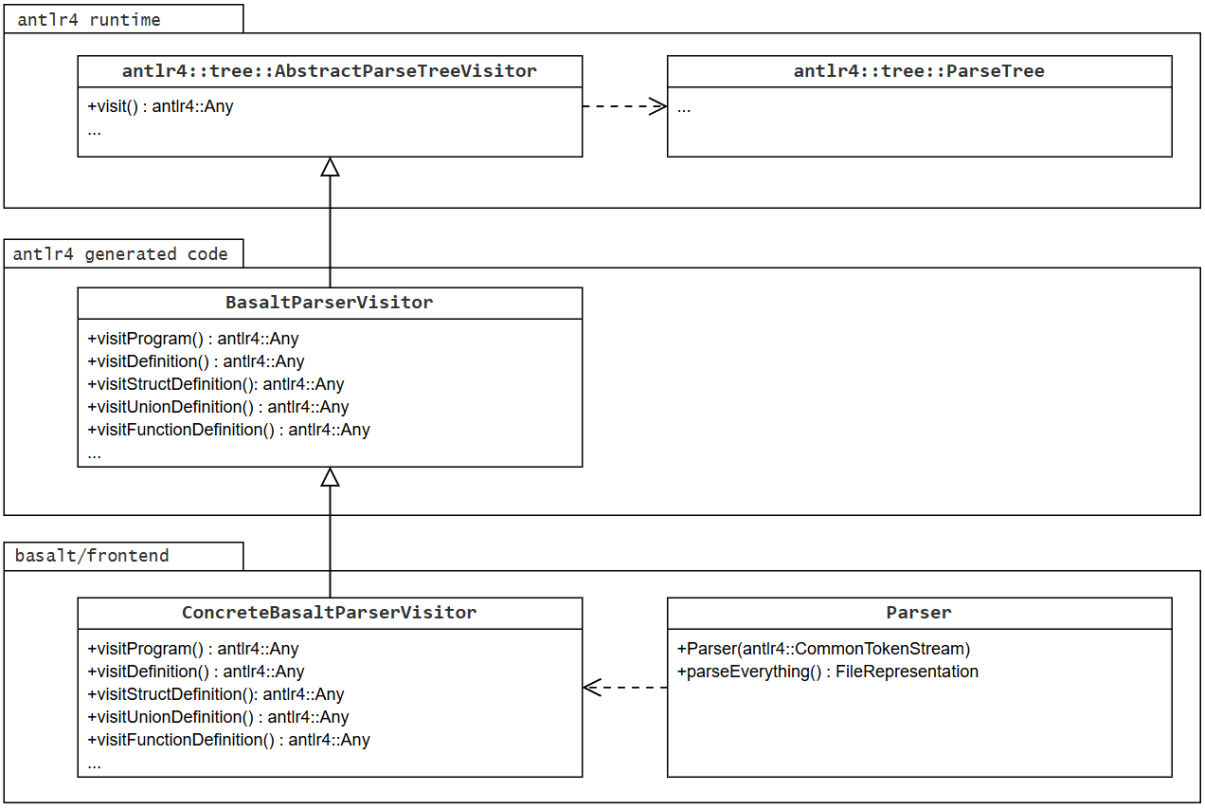


Figura 14: UML-Class-diagram del parser con ANTLR4

Il risultato di ogni chiamata ai vari metodi di `ConcreteBasaltParserVisitor` è un oggetto di tipo `antlr4::Any`, il quale è un tipo di dato fornito da ANTLR per modellare il risultato di una generica visita al parse tree.

Nel caso specifico, il risultato di ogni metodo di `ConcreteBasaltParserVisitor` è un'istanza di una classe definita nella codebase di Basalt, la quale rappresenta un'entità del linguaggio.

Utilizzando il metodo `parseProgram` il risultato sarà un'istanza di `FileRepresentation`, il quale codifica l'intero contenuto di un file sorgente.

Come input del costruttore della classe `Parser`, la quale è un wrapper su `ConcreteBasaltParserVisitor`, viene passato un oggetto di tipo `antlr4::CommonTokenStream`, il quale rappresenta il contenuto di un file sorgente.

Per ottenere un oggetto di tipo `antlr4::CommonTokenStream` è necessario utilizzare la classe `Tokenizer`, la quale è a sua volta un wrapper sulla classe autogenerata `BasaltLexer` (l'utilizzo dei wrapper serve ad esporre la stessa API della repository principale e consentire una migliore integrazione tra le due).

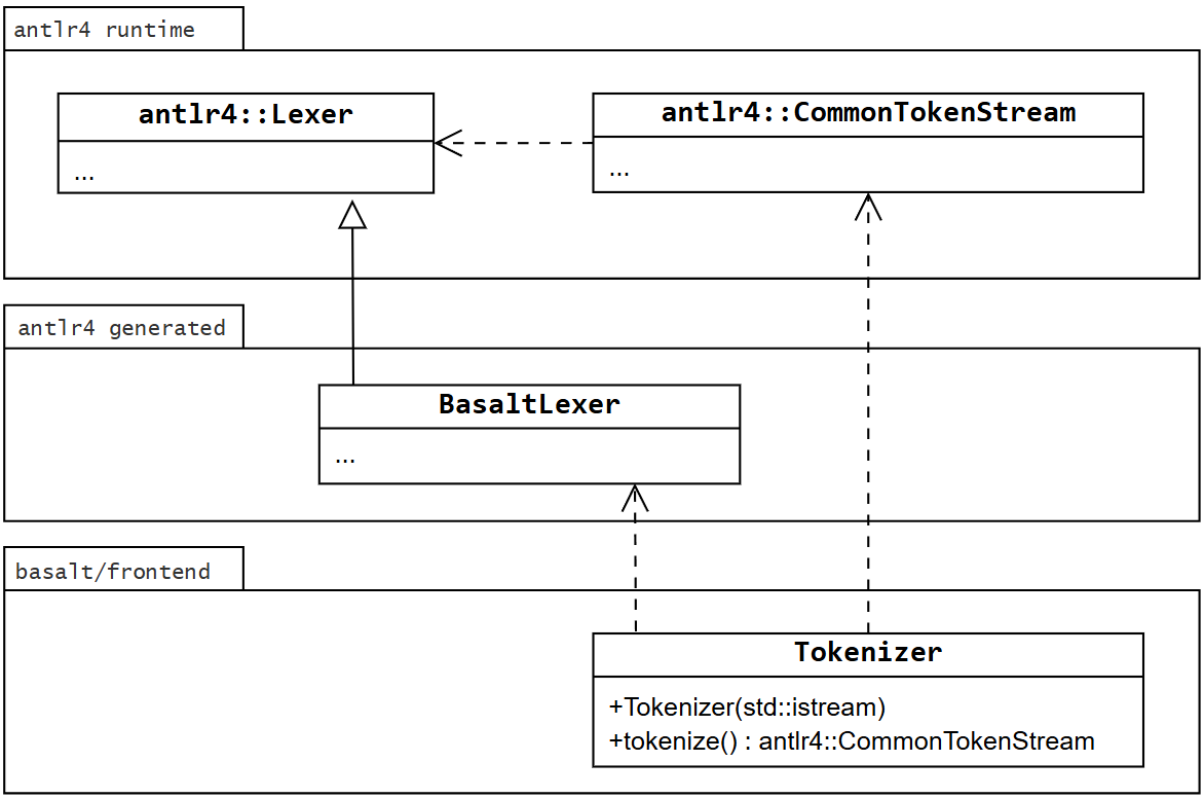


Figura 15: UML-Class-diagram del parser con ANTLR4

2.5.2 Tokenizzazione nella repository principale

Nella repository principale il tokenizer è stato realizzato su misura, senza l'uso di alcuna dipendenza esterna.

La classe `Tokenizer` espone due costruttori, uno per la costruzione a partire da un file ed uno per la costruzione a partire da una stringa , utilizzato durante il testing (in C++ è possibile trattare entrambi i casi in modo molto simile grazie alla classe `std::istream` che astrae sui dettagli implementativi sottostanti).

Il metodo `tokenize()` della classe tokenizer si occupa di effettuare la tokenizzazione dell'intero sorgente. È possibile schematizzare il funzionamento del metodo in pseudocodice come segue:

```
Tokenizer.tokenize():
    tokens = { }
    for line in source_code:
        char_index = 0
        while char_index < length(line):
            token, success = extract(line, &char_index)
            if (success):
                tokens.append(token)
    ensure_multiline_comments_closed()
    return tokens
```

Il metodo `tokenize()` si occupa di estrarre un singolo token per volta dal sorgente, esso è concretamente implementato come un dispatch sui vari metodi specifici per estrarre uno specifico tipo di token.

```
Tokenizer.extract(line, char_index):
    handle_multiline_comments(line, &char_index)
    handle_simple_comments(line, &char_index)
    first_char = line[char_index]
    tok = null
    switch(first_char):
        case '"' "'" '`': tok = extract_string(line, &char_index)
        case '0' ... '9': tok = extract_number(line, &char_index)
        case 'a' ... 'z': tok = extract_identifier(line, &char_index)
        case 'A' ... 'Z': tok = extract_typename(line, &char_index)
        case '+' ... '=': tok = extract_symbol(line, &char_index)
    return tok
```

I singoli metodi `extract_string`, `extract_number`, `extract_identifier`, `extract_typename`, `extract_symbol`, verificano se il testo della riga corrente, partendo dalla posizione `char_index`, coincida con quanto atteso. In caso di successo, essi restituiscono il token corrispondente alla porzione di testo che è stata riconosciuta e si occupano di avanzare l'indice fino al primo carattere non riconosciuto.

Il tokenizer mantiene traccia dei commenti multi-riga mediante uno stack. Esso viene popolato con i token relativi alle aperture dei commenti multi-riga (tali token sono salvati solo temporaneamente ai fini di error checking).

Ad ogni apertura di un commento multiriga, esso viene tokenizzato e salvato nello stack, mentre ad ogni chiusura di un commento multi-riga, si effettua un pop dallo stack.

```
Tokenizer.handle_multiline_comments(line, char_index):
    maybe_open_comment = line.substring(char_index, 2)
    if (maybe_open_comment == "/*"):
        comments_stack.push(make_token(line, char_index))
        char_index = char_index + 2
    while comments_stack.size() > 0 AND char_index < length(line):
        maybe_comment_seq = line.substring(char_index, 2)
        switch(maybe_comment_seq):
            case "/*":
                comments_stack.push(make_token(line, char_index))
                char_index = char_index + 2
            case "*/":
                comments_stack.pop()
                char_index = char_index + 2
            default:
                char_index = char_index + 1
```

Se al termine della tokenizzazione lo stack non dovesse essere vuoto, ciò significherebbe che almeno un commento multi-riga non è stato chiuso e i token che sono stati salvati nello stack potranno essere usati per fornire errori significativi in console.

Per quanto riguarda la gestione dei commenti su singola riga, essi sono gestiti in modo molto semplice. Qualora si incontri la sequenza di caratteri corrispondente all'apertura di un commento a singola riga, l'indice corrispondente al carattere corrente viene aggiornato in modo da far sembrare che la riga sia stata già tokenizzata fino al termine.

```
Tokenizer.handle_simple_comments(line, &char_index):
    maybe_open_comment = line.substring(char_index, 2)
    if (maybe_open_comment == "//"):
        char_index = length(line)
```

2.5.3 Parsing nella repository principale

Nella repository principale il parser è stato realizzato su misura, senza l'uso di alcuna dipendenza esterna. In particolare, il parser di Basalt è un parser *LL(*)* implementato mediante *recursive descent*.

Con *recurisve descent* si intende una tecnica implementativa di realizzazione di parser *LL(*)* che prevede l'utilizzo di funzioni (o metodi) i quali si chiamano a vicenda fra loro generando un grafo ricorsivo di chiamate.

Ad ogni chiamata viene passato, in termini di argomenti, tutto il necessario per tenere traccia del token corrente e dei token successivi durante il parsing. Ogni funzione si occupa di processare un certo numero di token a partire dal token corrente in avanti.

Ad esempio, per il parsing delle espressioni, si segue un approccio simile a quanto di seguito illustrato in pseudocodice, al netto di doverose ed opportune semplificazioni:

```
Parser.parse_expression(tok):
    Expression expr = parse_terminal_expression(&tok)
    if tok.current == BINARY_OPERATOR:
        expr = BinaryOperator {
            expr,
            tok.current(),
            parse_expression(tok.next())
        }
    return expr

Parser.parse_terminal_expression(tok):
    Expression expr = null
    switch token_stream_iterator.type:
        case IDENTIFIER:      expr = parse_identifier(&tok)
        case INT_LITERAL:     expr = parse_int_literal(&tok)
        case FLOAT_LITERAL:   expr = parse_int_literal(&tok)
        case BOOL_LITERAL:    expr = parse_bool_literal(&tok)
        case STRING_LITERAL:  expr = parse_string_literal(&tok)
        case ARRAY_LITERAL:   expr = parse_array_literal(&tok)
        case PREFIX_OPERATOR: expr = parse_prefix_operator(&tok)
        case OPN_PARENTHESYS: expr = parse_wrapped_expr(&tok)
    return expr

Parser.parse_prefix_operator(tok):
    UnaryOperator prefix_op = {
        tok.current,
        parse_expression(tok.next())
    }
    return prefix_op
```

2.5.4 Implementazione dell'AST

L'output della fase di parsing è un oggetto di tipo `FileRepresentation`. Esso è sostanzialmente la radice dell'AST relativo ad un singolo file sorgente, ed è composto da una collezione di radici di sotto-alberi corrispondenti alle definizioni contenute nel file.

```
struct FileRepresentation {

    struct Metadata {
        string filename;
        string packagename;
        vector<string> imports;
    };

    Metadata file_metadata;
    vector<TypeDefinition> type_defs;
    vector<FunctionDefinition> func_defs;
};
```

La classe `TypeDefinition` rappresenta una definizione di tipo, ed è super-tipo di tutti i nodi dell'AST che rappresentano definizioni di tipo.

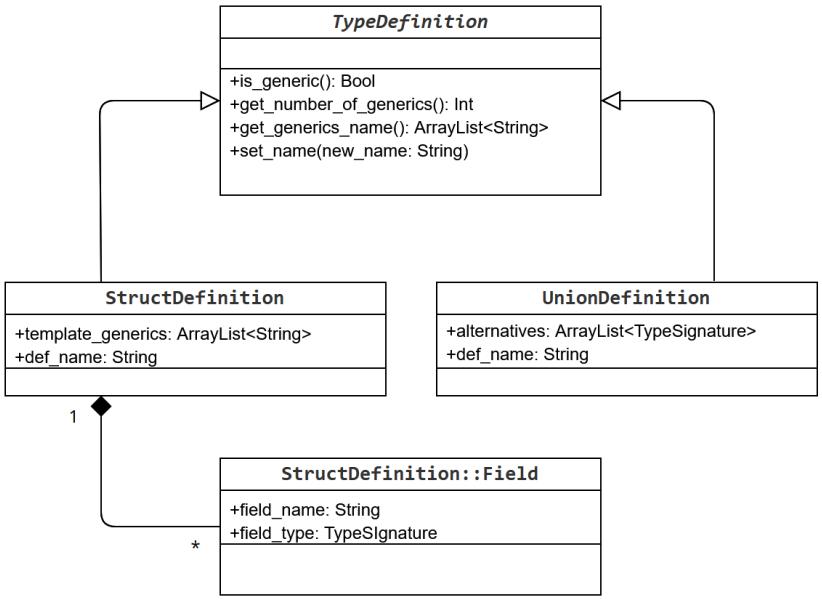


Figura 16: UML class diagram dei nodi dell'AST che rappresentano le definizioni di tipo

Di seguito segue l'UML class diagram relativo alle classi che modellano le espressioni in forma di AST. Si tenga presente che anche la repository basata su ANTLR utilizza la medesima rappresentazione, e si utilizza il visitor autogenerato da ANTLR per navigare la rappresentazione di ANTLR e tradurla opportunamente.

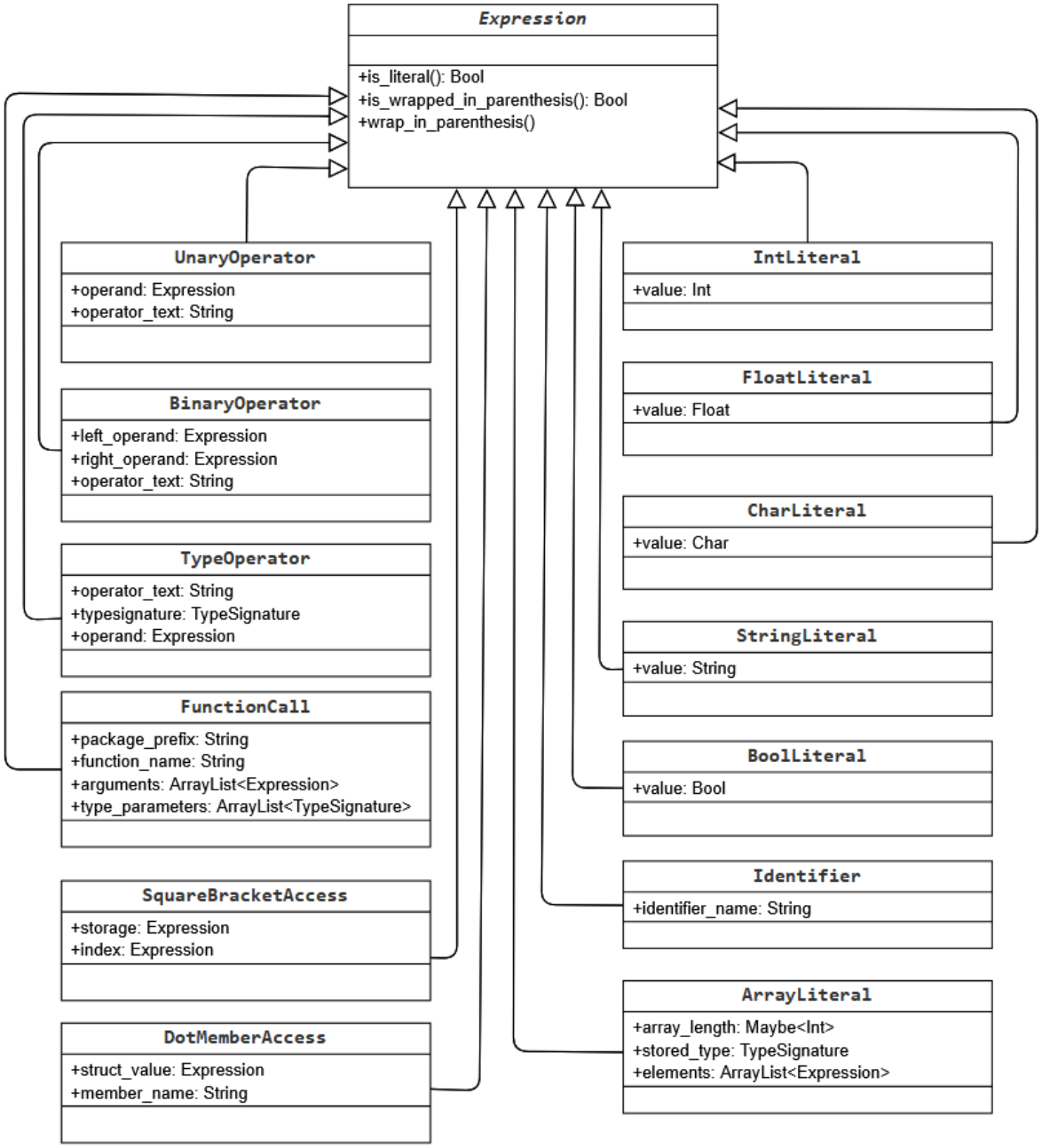


Figura 17: UML class diagram dei nodi dell'AST che rappresentano espressioni

Di seguito segue l'UML class diagram relativo alle classi che modellano gli statement in forma di AST. Così come già detto, anche la repository basata su ANTLR utilizza la medesima rappresentazione, e si utilizza il visitor autogenerato da ANTLR per navigare la rappresentazione di ANTLR e tradurla opportunamente.

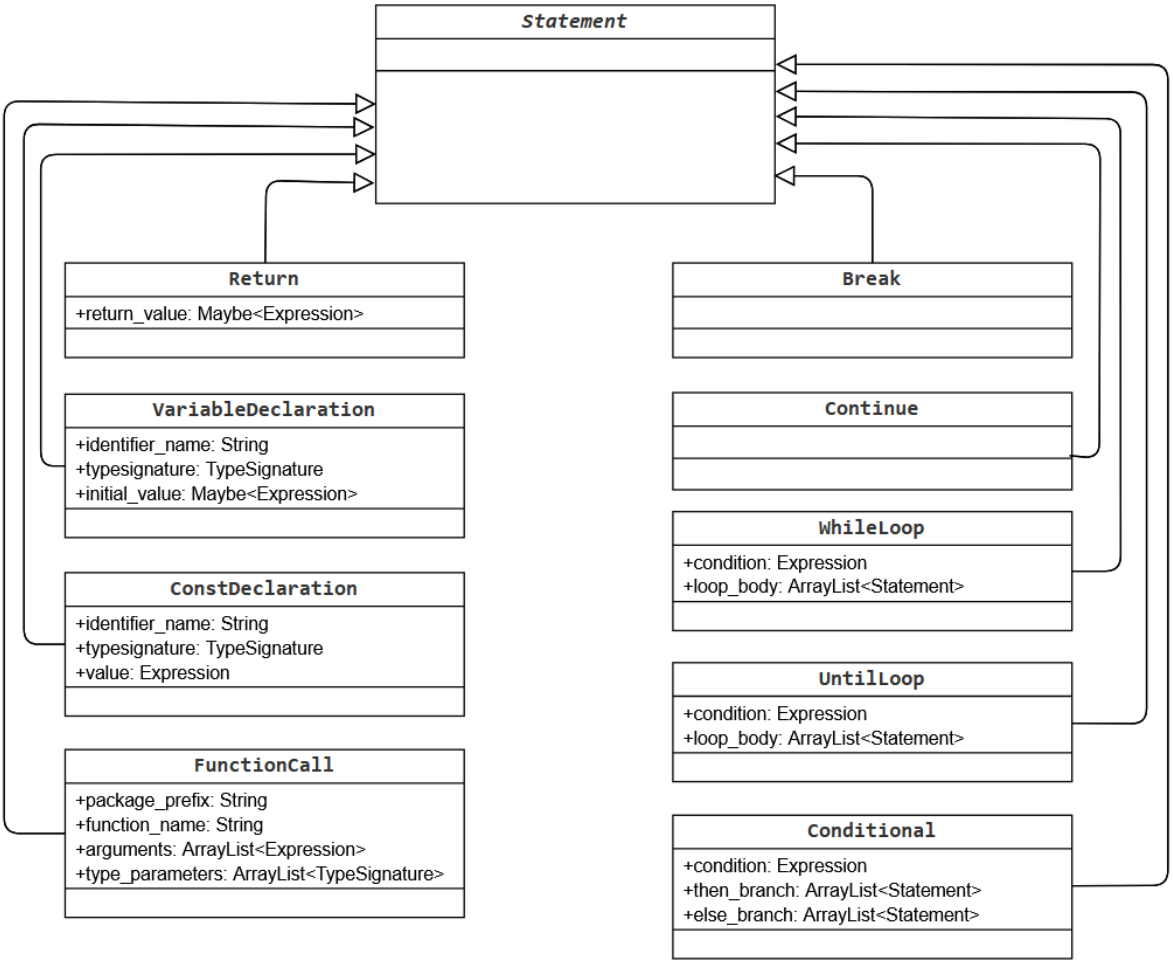


Figura 18: UML class diagram dei nodi dell'AST che rappresentano statement

Si può osservare che la classe `FunctionCall` è sia un sottotipo di `Expression` sia di `Statement`. Questo è dovuto al fatto che una chiamata di funzione può essere usata sia come espressione, se restituisce un valore, sia come statement, in caso contrario.

Si ricordi infatti che in C++ esiste l'ereditarietà multipla, e quindi è possibile che un una classe abbia più di un super-tipo anche nei casi in cui si usa l'ereditarietà.

Di seguito segue l'UML class diagram relativo alle classi che modellano type-signatures e definizioni di tipo in forma di AST. Così come già detto, anche la repository basata su ANTLR utilizza la medesima rappresentazione, e si utilizza il visitor autogenerato da ANTLR per navigare la rappresentazione di ANTLR e tradurla opportunamente.

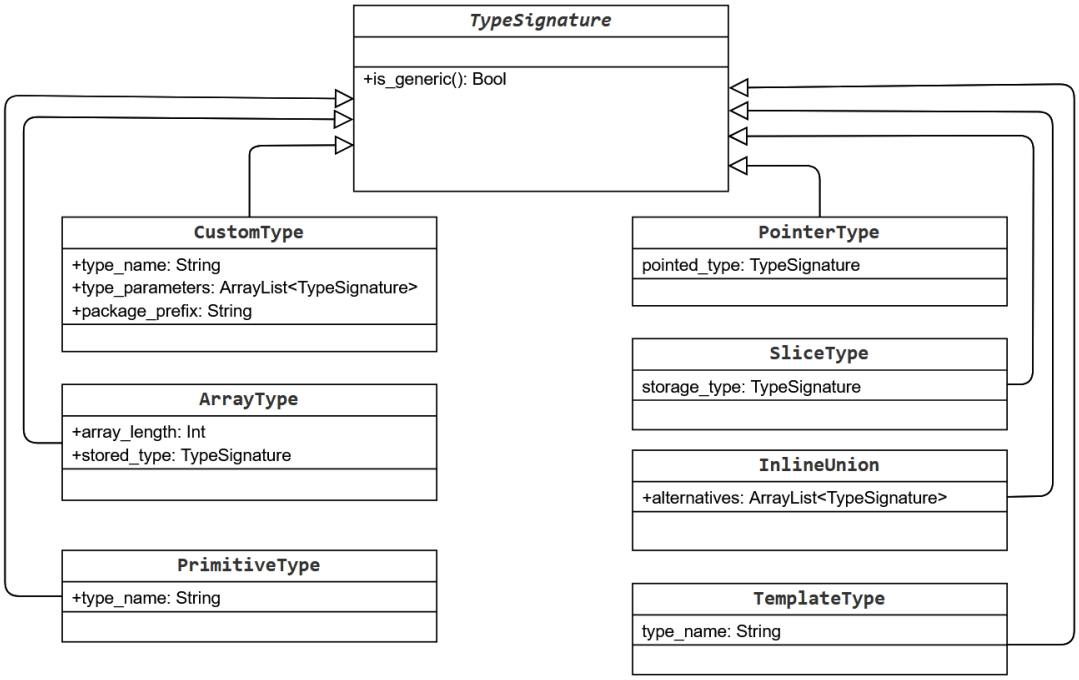


Figura 19: UML class diagram dei nodi dell'AST che rappresentano le type-signature

2.5.5 Tracciamento delle coordinate

Ogni classe dell'AST è sotto-tipo propriamente detto (via ereditarietà) della classe `CoordinatesAwareEntity`, la quale contiene informazioni utilizzabili per fornire messaggi di errore contestualizzati e significativi.

Tali informazioni sono dette coordinate, e consentono di tracciare la posizione dell'entità (Tipo, Funzione, Statement, Espressione) all'interno del file sorgente.

```
struct CoordinatesAwareEntity {

    string filename;
    int line_number;
    int tok_number;
    int char_pos;

    /* ... */
};
```

2.5.6 Precedenza degli operatori

Sia che l'AST venga generato da ANTLR, sia che venga generato manualmente, è necessario che l'AST rispetti la precedenza degli operatori. In fase di costruzione dell'AST, gli operatori vengono inseriti come nodi dell'albero seguendo il principio di fondo che un operando seguito da un operatore, a sua volta seguito da un'espressione, è un'espressione.

Subito dopo l'inserimento dell'espressione come figlio destro del nodo operatore, è necessario verificare che la precedenza dell'operatore appena inserito sia maggiore o uguale alla precedenza dell'operatore che si trova nel nodo padre. In caso contrario, è necessario ruotare l'albero in modo che l'operatore con la precedenza maggiore si trovi in cima all'albero.

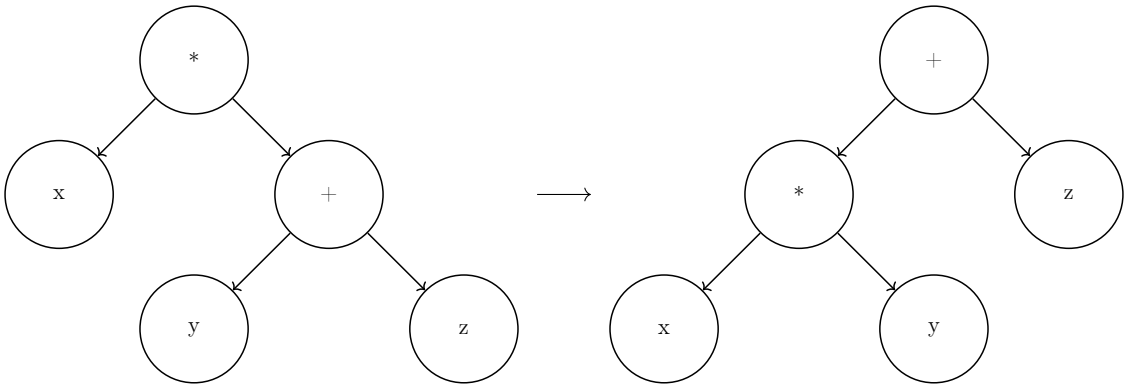


Figura 20: Esempio di generico AST formato da operatori binari prima e dopo una rotazione

Una situazione simile si verifica anche per gli operatori unari, come ad esempio l'operatore di negazione logica (!). Di seguito è riportato uno scenario di rotazione riguardante operatori logici binari ed unari.

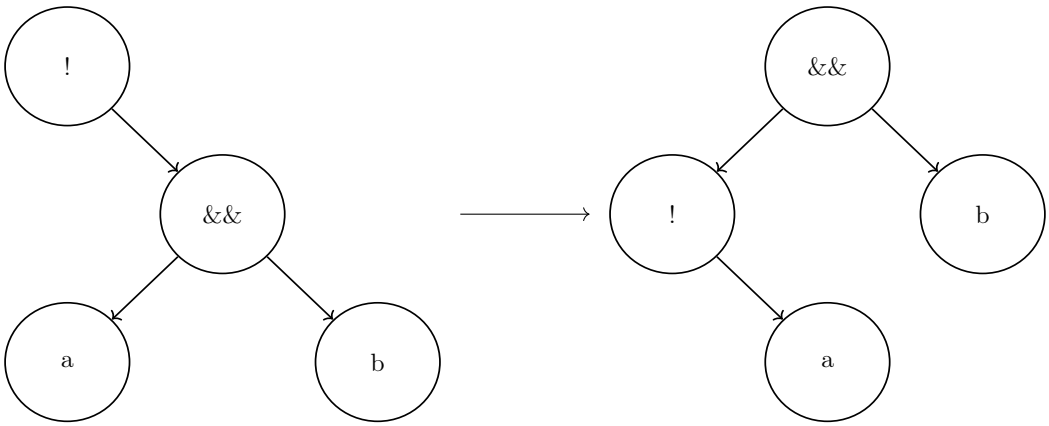


Figura 21: Esempio di generico AST formato da operatori unari e binari prima e dopo una rotazione

2.6 Indicizzazione: Symbol tables

Con *indicizzazione* ci si riferisce in generale al processo di creazione delle symbol-tables e della loro gestione. In questo capitolo si affronterà il tema della logica interna del compilatore Basalt che passa per l'indicizzazione di tutte le definizioni di tutti i file sorgente e che si conclude nell'analisi statica del programma indicizzato.

2.6.1 Merge degli output di parsing dei vari file sorgente

Basalt supporta la compilazione multi-sorgente, ovvero è possibile compilare più di un file per volta. Ciò significa che vi saranno molteplici oggetti di tipo `FileRepresentation` da dover unificare.

In generale, al termine della fase di indicizzazione sarà prodotto un unico oggetto di tipo `ProgramRepresentation` il quale rappresenta un layer di astrazione sulle varie symbol-table nonchè su alcune utility di gestione del filesystem che saranno necessarie nelle successive fasi della compilazione. Di seguito alcuni fra i metodi più significativi della classe `ProgramRepresentation`:

METODI	***BREVE DOCUMENTAZIONE***
`retrieve_type_definition`	Prende in input un oggetto di tipo `CustomType`, sottotipo di `TypeSignature`, restituisce la `TypeDefinition`, eventualmente già reificata
`resolve_function_call`	Prende in input una `FunctionCall` ed uno `ScopeContext`, restituisce un oggetto di tipo `CallableCodeBlock`, facendo overloading resolution e reificando il risultato se necessario
`resolve_expression_type`	Prende in input una `Expression` ed uno `ScopeContext`, restituisce il tipo dell'espressione in forma di `TypeSignature` oppure `null` se il contesto non fornisce informazione a sufficienza.
`validate_assignment`	Restituisce `true` se il tipo passato come primo argomento è assegnabile al tipo passato come secondo, `false` altrimenti

Tabella 7: Metodi più significativi della classe `ProgramRepresentation`

La classe `ProgramRepresentation` è nient'altro che una facciata per tutta la logica di indexing, essa espone una API semplificata alle altre classi che si occupano di analisi statica e/o di traduzione in IR.

Il motivo di questa scelta è che durante lo sviluppo è capitato più volte di dover cambiare la struttura interna delle symbol-tables, e avere un layer di astrazione in più ha di molto facilitato il refactoring.

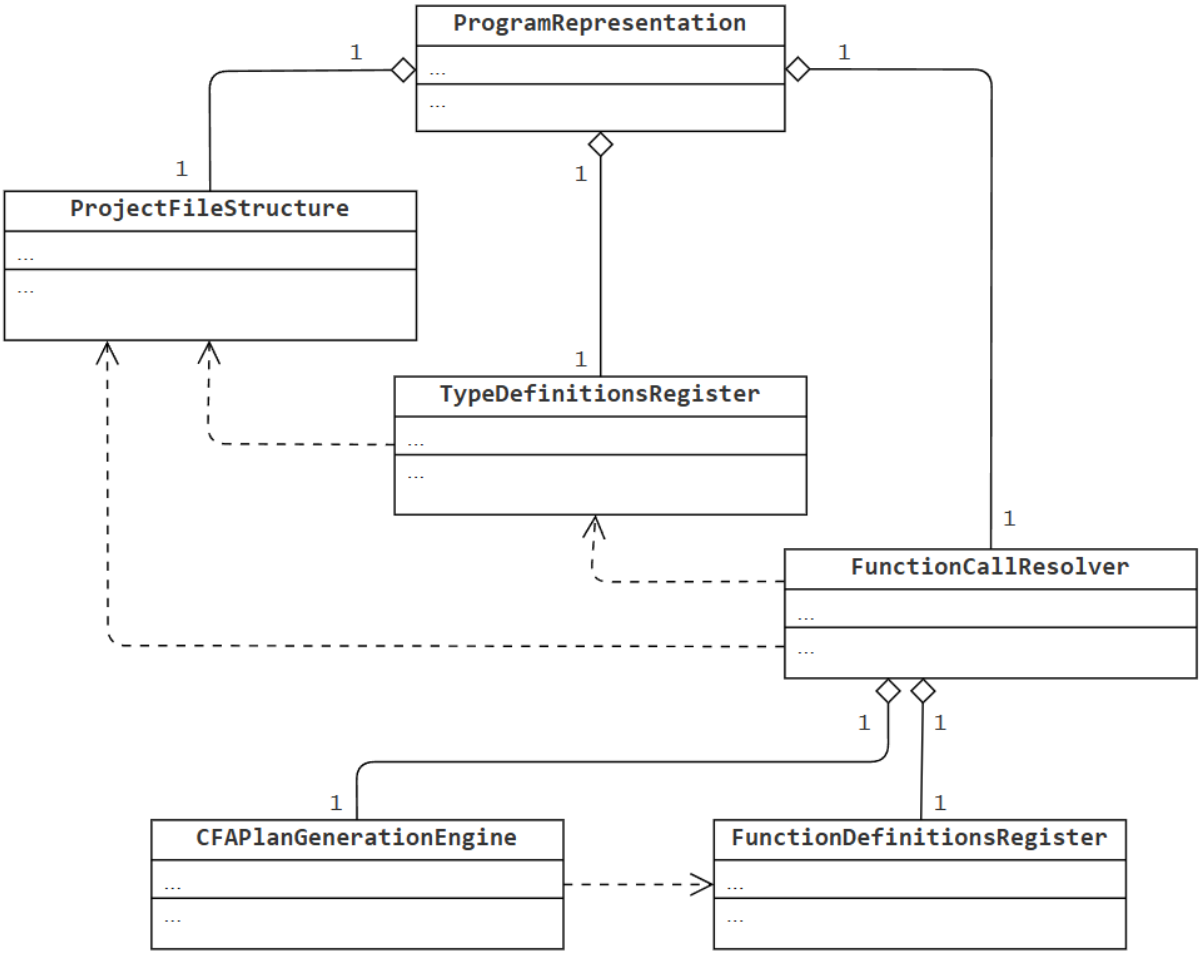


Figura 22: Diagramma UML semplificato delle relazioni tra le symbol-tables

La classe `FunctionCallResolver` è un layer di astrazione ulteriore rispetto ai diversi processi di risoluzione di una chiamata a funzione (e.g. risoluzione di una chiamata semplice, risoluzione di una chiamata che fa uso di *Common Features Adoption*).

Dato che le classi che si occupano della gestione del typesystem sono strettamente legate alla classe `TypeDefinitionsRegister`, anche l'interazione con queste ultime è interamente mediata dalla classe `ProgramRepresentation`.

2.6.2 Costruzione della tabella dei file/package

La tabella dei file e dei package è una struttura dati che traccia le relazioni fra diversi file sorgenti. Essa tiene traccia dei package importati da ogni file e dei file contenuti in ogni package.

Internamente essa è implementata sotto il nome di `ProjectFileStructure` ed è un wrapper su delle hash-map che permettono di effettuare ricerche in tempo costante.

```
class ProjectFileStructure {

    public:
        ProjectFileStructure() = default;
        ProjectFileStructure(vector<FileRepresentation>&);

        /* public exposed API */

    private:
        /* other internal data-structures */

        list<string> package_names;
        unordered_map<string, string> package_name_by_file_name;
        unordered_map<string, list<FileRepresentation>> files_by_pkg;
        unordered_map<string, vector<string>> imports_by_file;
};
```

Tale classe possiede i metodi necessari ad iterare sui file di un package, a trovare il package di un file, a trovare i file importati da un file e soprattutto a registrarli (e.g. indicizzarli, conservarli all'interno della struttura dati).

I restanti metodi della classe sono di supporto e permettono di iterare sui file e sui package, o di ottenere informazioni specifiche per un dato file o package.

Durante la compilazione di un programma Basalt, viene costruita una ed una sola istanza di `ProjectFileStructure` che sarà passata come parametro di costruttore alle restanti tabelle e strutture dati che necessitano le informazioni che essa codifica.

L'utilizzo della lista `package_names` permette di avere una collezione da poter iterare anche durante la modifica, ciò consente di poter alterare le mappe sottostanti senza invalidare l'iterazione.

In C++ infatti, non è possibile modificare una hash-map durante l'iterazione, in quanto l'iteratore potrebbe diventare invalido (ciò non avviene con le liste).

2.6.3 Costruzione della tabella dei tipi

La tabella dei tipi è una struttura dati che traccia le definizioni di tutti i tipi e rende possibile risalire ad esse partendo da un oggetto di tipo `TypeSignature`.

```
class TypeDefinitionsRegister {

    public:
        TypeDefinitionsRegister(ProjectFileStructure&);

        void store_type_definition(TypeDefinition&);
        TypeDefinition retrieve_type_definition(TypeSignature&);

        /* public exposed API */

    protected:
        /* other internal methods */

    private:
        list<string> type_definitions_ids;
        unordered_map<string, TypeDefinition> type_definitions;
        ProjectFileStructure& project_file_structure;
};
```

Internamente, la tabella dei tipi è implementata come una mappa che associa ad ogni nome di tipo, completamente qualificato, la definizione del tipo corrispondente. Inoltre, viene mantenuta una lista dei nomi di tipo per consentire l'iterazione con modifica in modo analogo a quanto visto nel paragrafo precedente.

Per registrare una definizione di tipo all'interno della tabella, viene generata una stringa univoca nella forma `<package>::<type_name>::<N>` (dove `<N>` è il numero di parametri formali di tipo) e viene usata come chiave nella mappa per l'inserimento della definizione. In fase di recupero, la tabella dei tipi permette di ottenere la definizione di un tipo a partire da una `TypeSignature`. Ciò avviene mediante il recupero diretto della definizione associata alla chiave generata a partire dal nome qualificato del tipo, se tale tipo è stato utilizzando specificandone il package di provenienza (e.g. `math::Complex`).

Se il package di provenienza non è stato specificato, si tenterà prima con il package a cui appartiene il file in cui è stata trovata la `TypeSignature` e, in caso di fallimento, si proverà con tutti i package importati da tale file.

Risalire al package in cui si trova il file che contiene la `TypeSignature`, così come i package da esso importati, è un'operazione resa possibile dalla classe `ProjectFileStructure` approfondita precedentemente.

È possibile schematizzare il processo di recupero della definizione di un tipo in pseudocodice come segue (si assuma che `get_key` restituisca la chiave univoca):

```
TypeDefinitionsRegister.retrieve_type_definition(type_signature):
    current_package = project_file_structure
        .get_package_containing_file(type_signature.file)
    if type_signature.package != nil:
        key = get_key(current_package, type_signature)
        return type_definitions[key]
    else:
        imports = project_file_structure
            .get_imports_by_file(type_signature.file)
        for pkg in List.concat(current_package, imports):
            key = get_key(current_package, type_signature)
            if type_definitions[key] != nil:
                return type_definitions[key]
    return nil
```

In fase di inserimento, si controlla che non vi siano duplicati. In caso di duplicati, viene sollevata un'eccezione la quale corrisponderà ad un errore a tempo di compilazione dal punto di vista dell'utente. Se il recupero di una definizione di tipo fallisce viene sollevata ugualmente un'eccezione.

Si noti come il formato scelto per le chiavi consente di indicizzare correttamente tipi apparentemente duplicati ma con un numero diverso di parametri formali, come ad esempio `Pair<T>` e `Pair<T,U>`.

Si tenga poi a mente che l'operazione di generazione della chiave è un'operazione esosa di risorse. Il costo computazionale maggiore deriva dalla ricerca del package corretto iterando su tutti i package importati dal file in cui è stata trovata la `TypeSignature` qualora essa non abbia un package specificato e/o non sia trovata all'interno del package corrente.

L'utilizzo di una cache per velocizzare il recupero delle definizioni di tipo è stato considerato, ma non è stato implementato in quanto non si è ritenuto necessario. Tuttavia sarebbe possibile implementare un sistema di cache per memorizzare come un dato tipo sprovvisto di package esplicitamente specificato sia stato risolto partendo da uno specifico file.

Tale ottimizzazione favorirebbe codebase con pochi file molto grandi, mentre non sarebbe granchè d'impatto su codebase con molti file piccoli, per le quali, anzi, sarebbe sfavorevole.

2.6.4 Costruzione della tabella delle funzioni

La tabella delle funzioni è una struttura dati che traccia le definizioni di tutte le funzioni e rende possibile risalire ad esse partendo da un oggetto di tipo `FunctionCall` e dai tipi concreti degli argomenti passati alla chiamata.

```
class FunctionDefinitionsRegister {

    public:
        FunctionDefinitionsRegister(
            ProjectFileStructure&,
            TypeDefinitionsRegister&
        );

        void store_function_definition(FunctionDefinition&);

        FunctionDefinition retrieve_function_definition(
            const FunctionCall& function_call,
            const vector<TypeSignature>& arg_types
        );

        /* public exposed API */

    protected:
        /* other internal methods */

    private:
        /* other internal data-structures */

        list<FunctionDefinition> function_definitions;
        unordered_map<string, OverloadSet> overload_sets;
        unordered_map<string, FunctionDefinition> fast_retrieve_cache;

        TypeDefinitionsRegister& type_definitions_register;
        ProjectFileStructure& project_file_structure;
};
```

Così come per le altre tabelle, anche la tabella delle funzioni tiene traccia di tutte le definizioni di funzioni presenti nel progetto usando una lista, così da potervi iterare anche quando la si sta modificando (ad esempio generando nuove definizioni tramite reificazione).

Per effettuare il recupero della definizione a funzione a partire da una chiamata, è necessario fornire anche i tipi concreti degli argomenti passati alla chiamata per poter effettuare overloading resolution basandosi proprio sui suddetti tipi. La classe `ProgramRepresentation` si occupa di generare tali tipi a partire da uno `ScopeContext`.

In fase di registrazione di una definizione di funzione, si aggiunge la stessa ad un `OverloadSet`, il quale è una collezione di definizioni di funzioni che condividono lo stesso nome, lo stesso package, lo stesso numero di argomenti e parametri formali di tipo.

Dato che si può chiamare una funzione anche senza specificare i parametri formali di tipo, è necessario inserire una funzione generica anche nell'overload set di quello in cui essa sarebbe stata inserita se non fosse stata generica.

Se la chiamata a funzione specifica il package in cui la funzione è definita, si effettua una ricerca diretta nell'overload set corrispondente (e.g. `console::println` specifica il package `console`).

In fase di recupero di una definizione di funzione per il quale il package non è stato esplicitamente specificato, si effettua una ricerca nell'overload set corrispondente prima usando il package relativo al file all'interno del quale è stata effettuata la chiamata, e poi cercando tra tutti i package importati da tale file.

```
FunctionDefinitionsRegister.retrieve_function_definition(func, types):
    current_package = project_file_structure
        .get_package_containing_file(func.file)
    if type_signature.package != nil:
        key = get_key(current_package, type_signature)
        return search_in_overload_set(key, func, types)
    else:
        imports = project_file_structure
            .get_imports_by_file(type_signature.file)
        for pkg in List.concat(current_package, imports):
            key = get_key(current_package, type_signature)
            func = search_in_overload_set(key, func, types)
            if func != nil:
                return func
    return nil
```

L'algoritmo di recupero di una definizione di funzione, sopra schematizzato, tratta gli overload set in maniera analoga a quanto visto per le definizioni di tipo e analizza un overload set per volta. Solo che in questo caso, sarà necessario anche analizzare ogni funzione all'interno di un dato overload set.

L'operazione di scelta del miglior candidato, ove possibile, da un dato overload set è detta *overloading resolution*.

Dato che tale operazione è *molto* esosa di risorse, si è deciso di implementare una cache per velocizzare il recupero delle definizioni di funzione. La cache è implementata come una mappa che associa ad una chiave di tipo stringa nella forma calcolata a partire dal nome del file in cui si trova una chiamata, il nome della funzione e i tipi concreti dei generics e degli argomenti della definzione stessa.

Durante la ricerca della definizione di una funzione all'interno di un overload set, si tiene traccia dei candidati migliori (più specifici) e, una volta terminata la ricerca, se è stato trovato un solo candidato, esso viene restituito.

In caso di più candidati viene generato un errore a tempo di compilazione per chiamata ambigua, mentre, nel caso di zero candidati, si prosegue la ricerca nell'overload set successivo.

Per ogni definizione di funzione dell'overload set si verifica se essa sia compatibile con i tipi concreti degli argomenti. Se e solo se la funzione risulta compatibile allora si genera un `FunctionSpecificityDescriptor` il quale viene comparato con il `FunctionSpecificityDescriptor` del miglior candidato trovato finora.

In caso di funzione ugualmente specifica, essa viene salvata tra i potenziali candidati. In caso di funzione più specifica, essa diventa il nuovo miglior candidato. In caso di funzione meno specifica, essa viene scartata.

```
FunctionDefinitionsRegister.search_in_overload_set(key, f, arg_types):
    best_matches = { }
    best_so_far = FunctionSpecificityDescriptor.worst()
    for func_def : overload_sets[key]:
        current_specificity = FunctionSpecificityDescriptor(func_def)
        if comparison_result.less_specific():
            continue
        comparison_result = current_specificity
            .compare_with(best_so_far);
        if !is_compatible(func_def, f, arg_types):
            continue
        best_maches_so_far.append(func_def);
        if comparison_result.more_specific():
            best_maches_so_far = { func_def };
            best_specificity_so_far = current_specificity;
    return extract_single_or_zero_best_match(best_matches);
```

Successivamente sarà dedicato l'opportuno spazio alla trattazione approfondita sia del sistema di scoring delle funzioni che all'istanziazione delle definizioni generiche (sia di tipi che di funzioni).

È opportuno precisare che lo pseudocodice mostrato è semplificato e non riflette esattamente l'implementazione reale, che invece si occupa anche di reificare le definizioni generiche ed aggiornare la cache mappando la definizione trovata con una opportuna chiave per il recupero veloce.

Anche il funzionamento del metodo `is_compatible` è particolarmente complesso e verrà trattato in dettaglio in seguito con il dovuto rigore formale.

2.6.5 Scoring degli overload

Durante il processo di *overloading resolution*, viene generato un `FunctionSpecificityDescriptor` per ogni funzione candidata. Questo oggetto contiene informazioni utili per individuare la funzione più specifica.

Una considerazione da fare è che due funzioni in Basalt possono essere comparate per specificità in senso assoluto, e non necessariamente in relazione al contesto in cui sono state chiamate. Questo semplifica di molto la logica di attribuzione dello score. Si ricordi:

- un overload non generico viene sempre preferito rispetto ad un overload generico, il numero di parametri di tipo non è rilevante
- un overload viene preferito ad un altro se i suoi parametri di tipo compaiono meno volte nei tipi dei suoi argomenti
- un overload viene preferito ad un altro se i suoi argomenti hanno tipi più complicati, ovvero hanno più parametri di tipo, oppure tali parametri di tipo sono a loro volta, ricorsivamente, più complicati
- un overload viene preferito ad un altro se tra i tipi dei suoi argomenti compaiono meno volte tipi definiti come union (anonime e non)
- un overload viene preferito ad un altro se il numero totale di casi coperti dalle union che compaiono tra i suoi argomenti è minore
- un overload viene preferito ad un altro se tra i tipi dei suoi argomenti e/o tra i parametri di tipo di questi ultimi vi sono meno conversioni di tipo

Come è possibile constatare dalle regole sopra elencate, la specificità di una funzione è un concetto che può essere valutato in modo assoluto, senza dover considerare il contesto in cui la funzione è stata chiamata.

In particolare, la valutazione delle conversioni di tipo è un'operazione che può essere effettuata in modo assoluto, in quanto, per un qualunque tipo `T`, il compilatore è a conoscenza del numero esatto di tipi diversi $\texttt{U}_1$, $\texttt{U}_2$, ..., $\texttt{U}_i$ tali che espressioni di quel tipo siano assegnabili a `T`. Questo numero è finito e noto a priori previa l'opportuna interrogazione della tabella dei tipi.

Quindi, in generale, è possibile ordinare tutte le funzioni di un overload set per specificità in modo assoluto. Poi, la/le più specifica/e tra queste potrebbero venire scartate per incompatibilità con i tipi concreti degli argomenti, difatti portando ad una corretta overloading resolution.

Un `FunctionSpecificityDescriptor` è quindi una strcut che contiene diversi indici corrispondenti a ciascuna delle regole sopra elencate, sul quale viene effettuato un confronto secondo il principio dell'ordinamento lessicografico.

2.6.6 Gestione ad alto livello della CFA

Avendo introdotto due molteplici tipi di codice eseguibile, ovvero le definizioni di funzioni e gli overload generati implicitamente tramite *Common Features Adoption* (CFA), allora è stato necessario introdurre un layer di astrazione rispetto ad essi tramite la classe `CallableCodeBlock`. Tale classe è super-tipo di entrambe secondo la definizione di super-tipo data nell'approfondimento dei design pattern in C++.

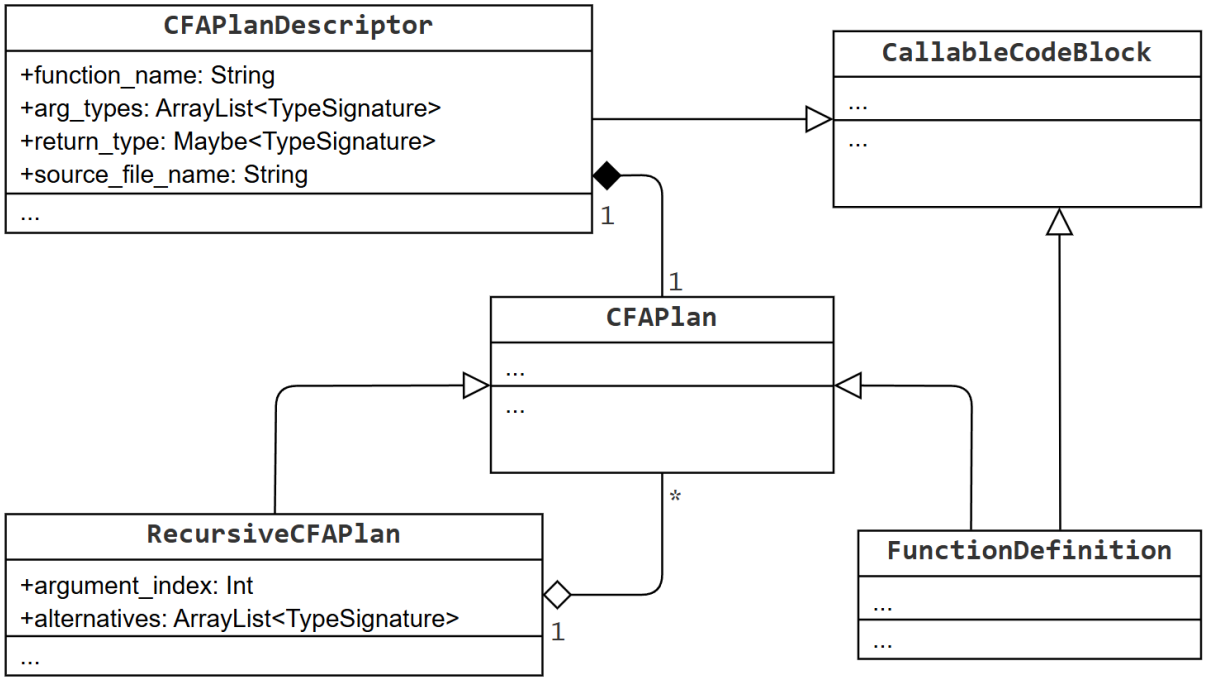


Figura 23: Diagramma UML semplificato delle relazioni tra le classi che modellano blocchi di codice eseguibile: definizioni di funzioni e overload CFA

Un `CFAPlanDescriptor` è un oggetto che modella la radice di un albero decisionale, di cui ogni foglia è una `FunctionDefinition`, mentre ogni nodo non-foglia è un `RecursiveCFAPlan` che codifica le possibili scelte durante il runtime-dispatch.

Si considerino infatti le seguenti definizioni di funzioni (saranno fornite solo le firme):

```
func max(x : Int,   y : Int)   -> Int   { /* ... */ }
func max(x : Float, y : Float) -> Float { /* ... */ }
func max(x : Float, y : Int)   -> Float { /* ... */ }
func max(x : Int,   y : Float) -> Float { /* ... */ }
```

Una chiamata alla funzione `max` con entrambi gli argomenti di tipo dichiarato `Int|Float` sarebbe risolta generando un'overload CFA, il quale sarebbe modellato come:

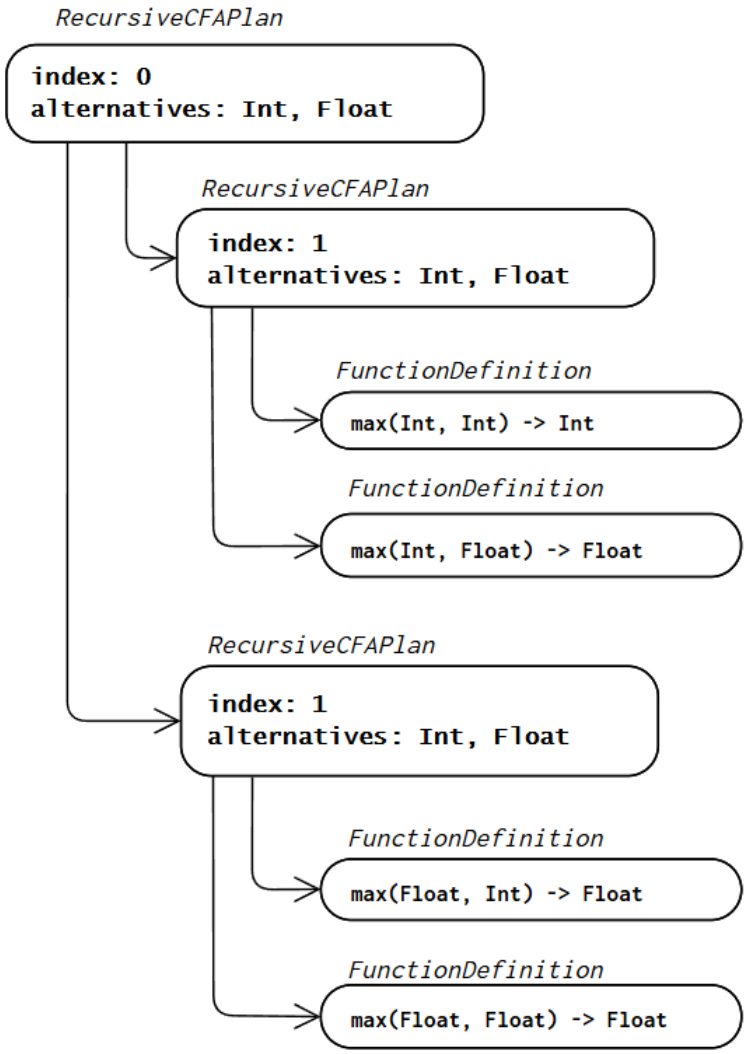


Figura 24: Memory-layout di un albero CFA per una chiamata alla funzione `max` con argomenti di tipo `Int|Float`

Tale albero rappresenta in tutto e per tutto un processo decisionale da effettuarsi a tempo di esecuzione dove per ogni nodo `RecursiveCFAPlan` ci si chiede quale sia il tipo concreto dell'argomento all'indice corrente, e si procede navigando sul figlio corrispondente. Giungere ad una foglia dell'albero durante la sua navigazione equivale a chiamare la funzione codificata da tale foglia.

La generazione di un `CFAPlanDescriptor` è responsabilità della classe `CFAPlanGenerationEngine`, la quale procede secondo l'algoritmo ora illustrato:

(dato che la generazione di un `CFAPlan` è esosa di risorse, la classe `CFAPlanGenerationEngine` utilizza una cache chiave/valore con chiavi stringhe costruite similmente a quanto visto per `FunctionDefinitionsRegister`):

```
CFAPlanGenerationEngine.generate_plan(func_call, arg_types):
    key = get_cache_fast_search_key(func_call, arg_types)
    if (cache[key] != nil):
        return cache[key]
    cfa_plan = generate_plan(
        function_call,
        arg_types,
        0 // <--- start from the first argument
    )
    cache[key] = cfa_plan
    return cfa_plan
```

```
CFAPlanGenerationEngine.generate_plan(func_call, arg_types, index):
    if (index >= function_call.arguments.size()):
        return nil
    fdef = search_function_definition(func_call, arg_types)
    if (fdef != nil):
        return fdef
    if (!function_call.arguments[index].is_union()):
        return generate_plan(
            function_call,
            arg_types,
            index + 1
        )
    plan = RecursiveCFAPlan()
    current = function_call.arguments[index].get_union()
    plan.alternatives = current.get_alternatives()
    for alternative in plan.alternatives:
        new_arg_types = arg_types
        new_arg_types[index] = alternative
        plan.children.append(
            generate_plan(
                function_call,
                new_arg_types,
                index + 1
            )
        )
```

2.7 Implementazione del typesystem

Il type-system è il componente del compilatore che si occupa di effettuare operazioni su definizioni di tipo e type-signature, di dedurre il tipo di una espressione e di controllare la compatibilità di due tipi rispetto all'operazione di assegnamento, eventualmente deducendo i vincoli di tipo necessari per i generics coinvolti e risolvendoli in un processo chiamato *type-inference*.

2.7.1 Reificazione dei generics

La reificazione delle definizioni generiche è responsabilità della classe `GenericsInstantiationEngine`. Di essa si crea un'istanza per ogni definizione che si desidera istanziare, costruendola a partire dalle regole di istanziazione, ovvero una struttura dati che tiene traccia di quale parametro attuale di tipo corrisponda a quale tipo generico. La reificazione avviene mediante *match-and-replace* (rimpiazzamento).

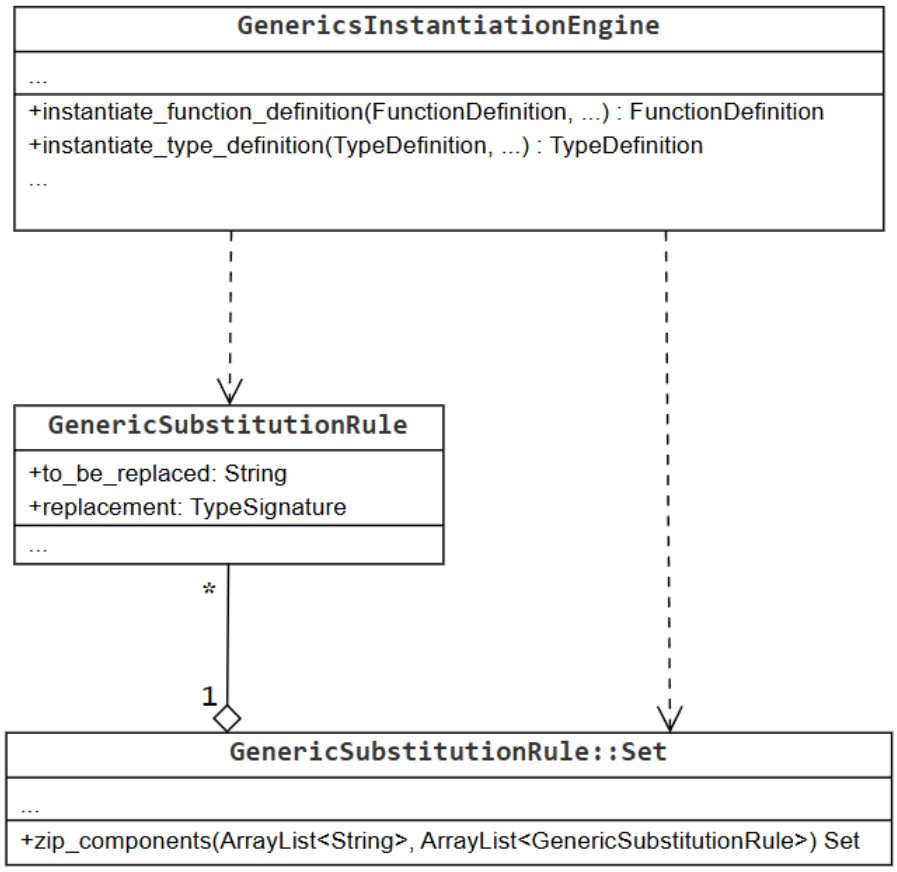


Figura 25: Diagramma UML delle classi che concorrono alla reificazione dei generics

L'implementazione di tale classe è abbastanza semplice, in quanto si limita a visitare ricorsivamente i nodi dell'AST dell'oggetto da istanziare, mediante visita in profondità (DFS), rimpiazzando i tipi generici con i tipi concreti qualora essa li incontra.

```
class GenericsInstantiationEngine {

    public:
        GenericsInstantiationEngine(GenericSubstitutionRule::Set&);

        TypeSignature instantiate_generic_type(TypeSignature&);

        Statement instantiate_generic_statement(Statement&);

        Expression instantiate_generic_expression(Expression&);

        TypeDefinition instantiate_generic_typedefinition(
            TypeDefinition& type_signature,
            string& new_type_name
        );

        FunctionDefinition::Ref instantiate_generic_function(
            FunctionDefinition& function_definition,
            string& new_function_name
        );

        FunctionDefinition::Ref instantiate_generic_function(
            FunctionDefinition& function_definition_ref,
            string& new_function_name
        );

        /* other methods */

    private:
        GenericSubstitutionRule::Set rules;
};
```

Per reificare (istanziare) una definizione, è necessario fornire un nuovo nome da dargli, in quanto due definizioni non possono avere lo stesso nome. All'interno della funzione generica istanziata si vanno a inserire i nomi completi dei tipi concreti al posto del numero di generics (si faccia riferimento al formato di indicizzazione delle definizioni nelle varie tabelle).

2.7.2 Deduzione del tipo di una espressione

La deduzione del tipo di un'espressione è responsabilità della classe `ExpressionTypeDeducer`. Di essa si crea un'istanza per ogni scope all'interno del quale si desidera analizzare le espressioni.

```
class ExpressionTypeDeducer {

    public:
        ExpressionTypeDeducer(
            TypeDefinitionsRegister&,
            FunctionDefinitionsRegister&,
            CommonFeatureAdoptionPlanGenerationEngine&,
            ProjectFileStructure&,
            ScopeContext&
        );

        std::optional<TypeSignature> deduce_expression_type(
            Expression& expression
        );

        std::vector<TypeSignature> deduce_arguments_types(
            FunctionCall& function_call
        );

        /* other methods */

    private:
        TypeDefinitionsRegister& type_definitions_register;
        FunctionDefinitionsRegister& function_definitions_register;
        CFAPlanGenerationEngine& cfa_plan_generation_engine;
        ProjectFileStructure& project_file_structure;
        ScopeContext& scope_context;
};
```

Il metodo `deduce_expression_type` è il metodo principale della classe, che si occupa di dedurre il tipo di un'espressione. Il risultato può essere `std::nullopt` (null) nel caso in cui la deduzione non sia possibile (e.g. dipendente da un tipo generico non ancora reificato).

Il metodo `deduce_arguments_types` è un metodo di supporto che si occupa di dedurre i tipi degli argomenti di una chiamata di funzione, al fine di poter effettuare l'overloading resolution. Nel caso in cui numero di tipi dedotti non coincida col numero di argomenti della funzione, ciò vuol dire che la deduzione di uno o più tipi non è stata possibile (e.g. dipendente da un tipo generico non ancora reificato).

La deduzione del tipo di un'espressione avviene mediante visita ricorsiva dell'AST dell'espressione stessa, attraverso una visita in profondità (DFS). Durante la visita si applicano le seguenti regole:

- **Literal**: il tipo di un letterale è noto a priori, e viene restituito direttamente.
- **Identifier**: il tipo di un identificatore è conservato all'interno dello `ScopeContext`.
- **Operatore unario**:
  - Se l'operatore è la negazione logica, il tipo dell'operatore è `Bool`
  - Se l'operatore è la dereferenziazione, si deduce che l'espressione è del tipo puntato dall'operando
  - Se l'operatore è la referenziazione (`&`), si deduce che l'espressione è un puntatore al tipo dell'operando
  - Negli altri casi, l'espressione è del tipo dell'operando
- **Operatore binario**:
  - Se l'operatore è un operatore aritmetico, si deduce il tipo dell'operando
  - Se l'operatore è un operatore di confronto, si deduce il tipo `Bool`
  - Se l'operatore è un operatore logico, si deduce il tipo `Bool`
  - Se l'operatore è l'operatore `is`, si deduce il tipo `Bool`
  - Se l'operatore è un cast, si deduce il tipo a cui si sta castando
- **Chiamata di funzione**: Si deduce il tipo di ritorno, previa overloading resolution
- **Accesso a membro**:
  - Se l'operando non è un puntatore, si deduce il tipo del membro richiesto
  - Se l'operando è un puntatore, si simula la sua dereferenziazione
  - Se l'operando è array/stringa/slice, si deduce il tipo `Int` per l'accesso a `.len`
- **Accesso a cella di array/slice**: Si deduce il tipo del generico elemento

2.7.3 Typechecking degli assegnamenti

Verificare che espressioni di tipo `T` siano assegnabili ad espressioni di tipo `U` è responsabilità della classe `AssignmentTypeChecker`. Tale classe espone una API estremamente minimale, ma al contempo fondamentale, composta di due soli metodi.

```
class AssignmentTypeChecker {

    public:
        AssignmentTypeChecker(
            TypeDefinitionsRegister& program_representation,
            ProjectFileStructure& project_file_structure
        );

        bool validate_assignment(
            const TypeSignature& source,
            const TypeSignature& dest,
            bool is_strict_mode
        );

        GenericSubstitutionRule::Set&
            get_generic_substitution_rules();

    private:
        /* other methods */

        TypeDefinitionsRegister& type_definitions_register;
        ProjectFileStructure& project_file_structure;
        set<string> strict_type_inference_deductions;
        GenericSubstitutionRule::Set generic_substitution_rules;
};
```

Il metodo `validate_assignment` è il metodo principale della classe, che si occupa di verificare che un'espressione di tipo `T` sia assegnabile ad un'espressione di tipo `U`. Esso prende tre argomenti, due dei quali sono le `TypeSignature` prese in esame mentre il terzo è una flag booleana che indica se il controllo debba essere effettuato in modalità *strict* (stringente, conservativa) o meno.

Tale metodo restituisce un valore booleano che indica se l'assegnamento è valido o meno. In caso l'assegnamento non possa essere controllato in quanto i tipi presi in esame sono generici e non ancora reificati, sarà ugualmente restituito `true`.

Presupposto che `TypeSignature` è supertipo di tutte le classi che modellano una type-signature nell'AST, e presupposto che quindi è necessario scoprire a tempo di esecuzione quale sia il tipo concreto degli oggetti il cui tipo dichiarato è `TypeSignature`, allora è chiaro come la chiamata al metodo `validate_assignment` debba ricorsivamente visitare l'AST delle due `TypeSignature`, mediante continue chiamate ricorsive a metodi interni della classe, per poi arrivare ad una ultima chiamata ad un metodo specifico per il caso concreto che si è verificato.

Alla luce di quanto detto è chiaro come la classe `AssignmentTypeChecker` abbia moltissimi metodi interni (privati), anche se ha solo due metodi pubblici. Ognuno di questi metodi interni implementa un grafo di chiamate altamente ricorsivo dove ad ogni chiamata ci si avvina allo scoprire i tipi concreti delle `TypeSignature`.

```
AssignmentTypeChecker.validate_assignment(source, dest, mode):
   switch dest:
      case TemplateType:  return validate_assignment_to_template(...)
      case CustomType:    return validate_assignment_to_custom(...)
      case PointerType:   return validate_assignment_to_pointer(...)
      case ArrayType:     return validate_assignment_to_array(...)
      case SliceType:     return validate_assignment_to_slice(...)
      case InlineUnion:   return validate_assignment_to_union(...)
      case PrimitiveType: return validate_assignment_to_primitive(...)

AssignmentTypeChecker.validate_assignment_to_pointer(source, ptr):
   if source.is_pointer():
       return validate_assignment(
          source.pointed_type(),
          ptr.pointed_type(),
          STRICT_MODE
       )
   else:
      return false
```

Nel precedente frammento di codice si intuisce l'utilizzo e lo scopo della flag relativa alla strict-mode. Tale flag è necessaria per segnalare che non si desidera restituire `true` qualora l'assegnamento comporti una conversione implicita di tipo.

Ad esempio, è possibile assegnare `Int` ad `Int | Float` ma non è possibile assegnare `List<Int>` a `List<Int|Float>` e nemmeno `#Int` a `#(Int|Float)`. La differenza fra questi scenari è che le chiamate al metodo `validate_assignment` fatte nei contesti appena illustrati vengono fatte in *strict-mode*.

Si consideri ad esempio il metodo `validate_assignment_to_union`, che si occupa di verificare se un'espressione di tipo `T` sia assegnabile ad un'espressione di tipo `U` dove `U` è un `InlineUnion` oppure un `CustomType` corrispondente ad una *named-union*:

```
AssignmentTypeChecker.validate_assignment_to_union(src, dest, mode):
    if mode == STRICT_MODE:
        return source.is_equal_to(dest)
    if src.is_equal_to(dest):
        return true
    for type in dest.alternatives():
        if validate_assignment(src, type, mode):
            return true
    for type in src.alternatives():
        if !validate_assignment(type, dest, mode):
            return false
    return true
```

Risulta evidente analizzando lo pseudocodice sopra riportato come il controllo di assegnabilità restituisca `true` per assegnamenti di tipo `Int` a `Int|Float` in modalità *non-strict*, e invece restituisca `false` in modalità *strict*.

In generale, valgono le seguenti affermazioni:

- `#T` è assegnabile a `#U` se e solo se `T` è assegnabile a `U` in modalità *strict*
- `$T` è assegnabile a `$U` se e solo se `T` è assegnabile a `U` in modalità *strict*
- `A<T>` è assegnabile ad `A<U>` se e solo se `T` è assegnabile a `U` in modalità *strict*
- `[]T` è assegnabile in modalità *non-strict* ad `[]U` se e solo se `T` è assegnabile a `U`
- `[]T` è assegnabile in modalità *strict* ad `[]U` se e solo se `T` e `U` sono identici
- `String` è assegnabile in modalità *non-strict* a `RawString`
- `A` è assegnabile in modalità *strict* a `B` se e solo se `A` e `B` sono identici
- la prima chiamata a `validate_assignment` è fatta in modalità *non-strict*

Valgono inoltre le regole di assegnabilità già trattate nella sezione dedicata al design del linguaggio al capitolo dedicato all'assegnabilità, in particolare, come confermato dal frammento di codice appena discusso, è possibile assegnare espressioni di tipo `SubType` ad espressioni di tipo `SuperType` in modalità *non-strict* se:

- `SubType` è identico a `SuperType`
- `SuperType` è una union e `SubType` è uno dei tipi che essa descrive
- `SubType` è una union e tutti i tipi da essa descritti sono assegnabili a `SuperType`

2.7.4 Algoritmo di Type-inference

Con *type-inference* si intende il processo che porta alla deduzione di uno o più tipi a partire dal contesto. In Basalt, l'unico scenario in cui si applica la type-inference è la deduzione dei parametri attuali di tipo di una chiamata a funzione.

Come detto in precedenza, durante la fase di *overloading-resolution*, il compilatore deve scegliere quale tra le funzioni contenute in un overload set è quella più specifica tra quelle *compatibili con la chiamata*.

Solo per le chiamate a funzione che non specificano i parametri attuali di tipo, ma che non sono compatibili con nessuna funzione non-generica, si tenta di dedurre i parametri attuali di tipo con la type-inference.

Durante l'atto di controllare che una definizione di funzione sia compatibile con una chiamata, viene controllata l'assegnabilità di tutti i tipi concreti degli argomenti rispetto ai tipi dichiarati dei parametri formali. Durante la validazione dell'assegnabilità, se il tipo a cui si desidera assegnare è generico, come *by-product* del controllo di assegnabilità, vengono dedotti i vincoli di tipo che legano quel tipo generico con il tipo concreto.

```
FunctionDefinitionsRegister.check_compatiblity(fdef, fcall, types):
   checker = new AssignmentTypeChecker(...)
   for actual, expected in arg_types(fdef, fcall):
      if !checker.validate_assignment(actual, expected, !STRICT_MODE):
         return (false, null)
   engine = new GenericsInstantiationEngine(
      checker.get_generic_substitution_rules() // inferred
   )
   instantiated = engine.instantiate(fdef, ...)
   return (true, instantiated)
```

Se la compatibilità è verificata, si restituisce non solo `true`, ma anche la definizione di funzione reificata grazie ai parametri attuali di tipo dedotti. Nel caso in cui i parametri attuali di tipo siano stati esplicitamente specificati, essi saranno passati al costruttore di `AssignmentTypeChecker` il quale li restituirà con il metodo `get_generic_substitution_rules` e l'algoritmo resterà invariato.

L'istanza di `AssignmentTypeChecker` in uso è specifica per la coppia definizione-chiamata in esame, essa tiene traccia internamente delle assunzioni che essa stessa effettua durante il type-checking e le restituisce al chiamante al termine. Tali assunzioni sono specifiche per l'istanza e quindi non hanno effetto sul controllo di compatibilità delle future definizioni.

I vincoli di assegnabilità che legano un parametro formale di tipo ad un tipo concreto sono dedotte in sede di controllo di validità dell'assegnabilità del tipo concreto al tipo generico.

```
AssignmentTypeChecker.validate_assignment_to_template(src, dst, mode):
    already_locked = false
    if strict_type_inference_deductions.contains(dst.type_name):
        already_locked = true
    if mode == STRICT_MODE:
        strict_type_inference_deductions.insert(dst.type_name);

    for rule in generic_substitution_rules:
        if rule.to_be_replaced != dst.type_name:
            continue
        if validate_assignment(src, rule.replacement, STRICT_MODE):
            return true;
        if validate_assignment(src, rule.replacement, !STRICT_MODE):
            return !already_locked
        if validate_assignment(rule.replacement, src, !STRICT_MODE):
            rule.replacement = src
            return !already_locked
        if rule.replacement.is_inline_union():
            rule.replacement.alternatives.append(src);
            return !already_locked
        rule.replacement = new InlineUnionType(
            rule.replacement, src
        );
        return !already_locked

    generic_substitution_rules.insert(dst.type_name, src)
    return !already_locked
```

In particolare, è possibile schematizzare ulteriormente l'algoritmo come segue:

- Se esiste già un plausibile candidato per il tipo generico, ed è possibile assegnare il tipo concreto a tale candidato in modalità *strict*, allora il candidato viene lasciato intatto e si restituirà `true` sia che ci si trovi in modalità *strict* che *non-strict*.
- Se non esiste un plausibile candidato per il tipo generico, il tipo che si desidera assegnare diventa esso stesso un candidato. Esso sarà scelto se e solo se l'assunzione non sarà affinata in seguito all'analisi dei rimanenti argomenti. In caso di *strict-mode*, l'assunzione sarà marcata come non-modificabile.
- In tutti gli altri casi si richiede di non essere in strict-mode, e si può mettere in discussione l'assunzione precedente rimpiazzandola con il tipo che si desidera assegnare, qualora tale modifica preservi i vincoli dedotti in precedenza, oppure con la union anonima del tipo concreto che si desidera assegnare con il precedente candidato.

Un dubbio che è lecito porsi è il seguente: È giusto che anche qualora in *strict-mode*, vi sia un modo modificare l'assunzione precedente basandosi sull'esito di un controllo di assegnabilità che non è stato condotto in *strict-mode*?

Si consideri la seguente definizione di funzione generica:

```
func contains<T>(target : T, list : List<T>) -> Bool {
    cursor : #Node = list.head;
    while (cursor != null) {
        if (utils::compare_eq(cursor.value, target)) {
            return true;
        }
        cursor = cursor.next;
    }
    return false;
}
```

Deve essere possibile chiamare la funzione `contains` con argomenti `Int` e `List<Int|Float>` affidandosi alla type-inference per dedurre il tipo di `T`. Al momento in cui si analizza l'assegnamento di `List<Int|Float>` a `List<T>` in modalità *non-strict*, il quale comporta il controllo dell'assegnamento di `Int|Float` a `T` in modalità *strict*, il tipo `T` avrà già un possibile candiato ovvero `Int`.

Tale assunzione è modificabile, ed è quindi giusto che venga modificata anche in *strict-mode*. Questo significa che la *strict-mode* impedisce l'assegnazione di un tipo ad un altro se questo avviene mediante conversione implicita di tipo, ma solo se il tipo destinzazione non è generico.

Per tipi di destinazione generici, la *strict-mode* impedisce solo la modifica futura dell'assunzione che si sta attualmente facendo.

La modifica dell'assunzione precedente è possibile se essa è modificabile e se il nuovo candidato non comporta la perdita di vincoli di tipo dedotti in precedenza. La non-perdita dei vincoli è garantita dal fatto che è stato già verificato che il nuovo candidato possa ricevere assegnamenti da parte del candidato precedente a patto di supportare le conversioni implicite di tipo, ciò significa che dunque è assolutamente corretto che tale controllo non avvenga in *strict-moode*.

Si ricordi che l'assegnabilità di un tipo ad un altro è una relazione transitiva, e quindi se è possibile assegnare ad un nuovo candidato il precedente, allora ciò implica che è possibile assegnarvi tutti i candidati che sono stati dedotti fin ora e che sono poi stati scartati. Questa semplice osservazione garantisce la correttezza dell'algoritmo.

2.8 Analisi statica

Con *analisi statica* si intende il processo di validazione e controllo del programma già indicizzato. Tale processo di controllo viene implementato mediante una serie di visitor che navigano l'AST e/o i grafi delle dipendenze.

2.8.1 Controllo di aciclicità delle dipendenze dirette fra tipi

Il primo controllo di correttezza ad essere stato implementato nella codebase è stato il controllo di aciclicità delle dipendenze dirette fra tipi.

Dato un qualunque tipo `T`, si considerano dipendenti direttamente da esso tutti i tipi che modellano array di `T`, tutti i tipi union definite a partire da esso o da suoi tipi dipendenti direttamente e tutti i tipo struct con membri di tipo `T` o suoi tipi dipendenti direttamente.

Se il grafo delle dipendenze dirette fosse ciclico, in particolare, se ad esempio il tipo `T` fosse direttamente dipendente da `U` e viceversa, ciò implicherebbe che non è possibile calcolare la dimensione in byte necessaria ad allocare in memoria oggetti ti tipo `T`.

Per garantire l'aciclicità del grafo, esso viene visitato in profondità (DFS) dalla classe `TypeDependencyNavigator`, la quale inzia la visita da una data definizione di tipo, e procede esplorando tutti i tipi da cui tale definizione è dipendente direttamente.

```
TypeDependencyNavigator.visit_type_definition(type_definition):
    union_id = union_definition.generate_union_id()
    if visited.contains(union_id):
        return
    if visiting.contains(union_id):
        raise new CyclicDependencyError(...);
    visiting.insert(union_id)
    for type in get_depnendencies(type_definitions):
        visit_typesignature(type, type_definitions.generics)
    visiting.remove(union_id)
    visited.insert(union_id)

TypeDependencyNavigator.visit_typesignature(typesignature, generics):
    if typesignature.is_generic():
        return
    else if typesignature.is_array():
        visit_typesignature(typesignature.stored_type, generics)
    else if typesignature.is_custom_type():
        retrieved_def = type_definitions_register
            .retrieve_type_definition(typesignature)
        visit_type_definition(type_definition)
```

2.8.2 Analisi dei legami post-assegnamento

Assegnando un espressione detta *sorgente* ad un'altra espressione detta *destinazione* è opportuno chiedersi se, manipolando la destinazione, non si possa alterare in qualche modo la sorgente.

Ad esempio, nel caso in cui sia sorgente che destinazione fossero puntatori, al termine dell'assegnamento, manipolando il puntatore destinazione si potrebbe accedere in scrittura all'area di memoria puntata dal puntatore sorgente.

In generale, ci si deve porre il problema di quando assegnare un'espressione sorgente espone tale espressione alla possibilità di essere indirettamente modificata, del tutto o in parte, mediante azioni di qualsiasi tipo compiute sulla destinazione.

La risposta a tale domanda è fornita dalla classe `BondInspector` e dall'algoritmo che essa implementa sottoforma di metodi mutuamente ricorsivi. Tale classe infatti implementa un metodo chiamato `does_the_type_of_this_expr_imply_a_bond` che, visitando in profondità (DFS) l'AST relativo ad una data typesignature, restituisce `true` se assegnare espressioni aventi tale tipo può portare a scenari di modifica indiretta.

```
BondInspector.does_the_type_of_this_expr_imply_a_bond(type):
    switch type:
        case InlineUnion:
            return does_this_inline_union_imply_a_bond(type)
        case CustomType:
            return does_this_custom_type_imply_a_bond(type)
        case ArrayType:
            return does_the_type_of_this_expr_imply_a_bond(
                type.stored_type
            )
        case PrimitiveType: return false
        case PointerType:   return true
        case SliceType:     return true
        case TemplateType:  return true

BondInspector.does_this_union_imply_a_bond(union):
    for alternative in union.alternatives:
        does_the_type_of_this_expr_imply_a_bond(alternative):
            return true
    return false
```

2.8.3 Analisi dell'osservabilità post-assegnamento

Assegnando un espressione detta *sorgente* ad un'altra espressione detta *destinazione* è opportuno chiedersi se sia realmente possibile osservare le modifiche apportate.

Ad esempio, nel caso in cui la destinazione fosse il valore restituito da una chiamata a funzione, non sarebbe possibile osservare alcuna modifica apportata alla destinazione, in quanto di essa non rimane più alcun riferimento ad assignment completato.

In generale, un assignment ad una espressione detta *destinazione* è considerato osservabile o non-osservabile in base in base a che tipologia di espressione sia la destinazione.

- **Operatori binari**: la loro modifica non è osservabile
- **Operatori unari (matematici)**: la loro modifica non è osservabile
- **Operatori unari (logici)**: la loro modifica non è osservabile
- **Operatori unari (su puntatori)**: la loro modifica è sempre osservabile
- **Cast**: osservabile se è osservabile ciò che si vuole castare
- **Accessi a membro**: la modifica è osservabile se lo sarebbe una modifica all'espressione a cui si accede oppure se l'espressione a cui si accede è un puntatore.
- **Accessi a cella**: la modifica è osservabile se lo sarebbe una modifica all'espressione a cui si accede oppure se l'espressione a cui si accede è una slice.
- **Chiamate a funzioni**: la loro modifica non è osservabile
- **Identificatori**: la loro modifica è sempre osservabile
- **Literal**: la loro modifica non è osservabile

A partire dalle seguenti regole, la classe `ObservabilityDeducer` implementa un metodo chiamato `is_expression_update_observable` che, visitando in profondità (DFS) l'AST relativo ad una data espressione, restituisce `true` se una ipotetica modifica a tale espressione sarebbe osservabile, `false` in caso contrario.

Si considerino ad esempio le seguenti definizioni (saranno fornite solo le firme delle funzioni per facilitare la lettura):

```
struct S {
    ptr : #S;
    val : Int;
}

func f() -> S {
    /* ... */
}

func g() -> #S {
    /* ... */
}

var s : S = /* ... */;
```

Date le definizioni appena fornite, si deducono i seguenti verdetti di osservabilità:

```
f() = s;
// Non-osservabile

f().val = 10;
// Non-osservabile

f().ptr.val = 10;
// Osservabile

g() = &s;
// Non-osservabile

g().val = 10;
// Osservabile

g().ptr.val = 10;
// Osservabile

#g() = s;
// Osservabile
```

2.8.4 Implementazione dell'immutabilità

Controllare che all'interno dell'intero programma non vi siano violazioni dei vincoli di immutabilità è responsabilità della classe `ImmutabilityConstraintValidator`. Tale classe visita l'AST relativo alle varie definizioni di funzione in profondità (DFS), per poi controllare che gli assegnamenti non violino tali vincoli.

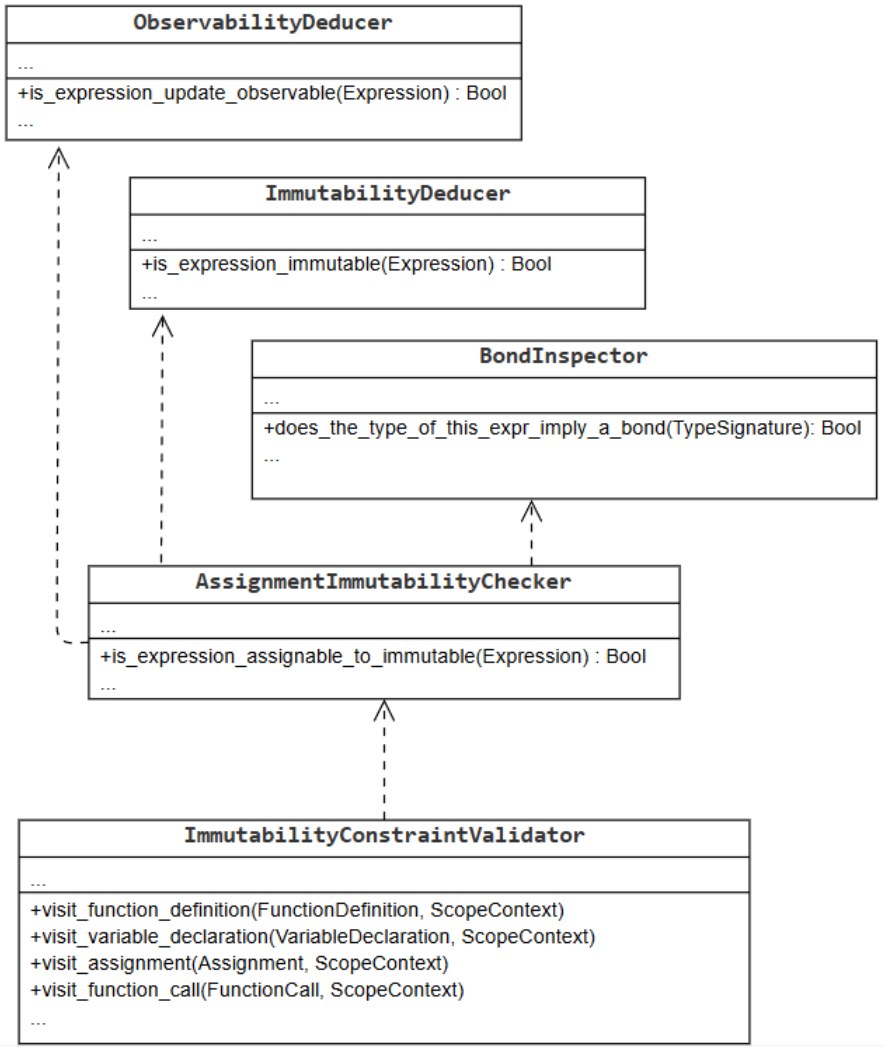


Figura 26: Diagramma UML delle classi che implementano i controlli di immutabilità

La classe `ImmutabilityConstraintValidator` verifica non solo la correttezza degli assegnamenti espliciti, ma anche la correttezza degli assegnamenti impliciti, come ad esempio l'assegnamento di un argomento di funzione ad un parametro formale e l'assegnamento del valore iniziale di una variabile alla suddetta variabile.

La classe `ImmutabilityDeducer` è una classe di appoggio con un singolo metodo pubblico. Tale metodo è `is_expression_immutable` e serve a navigare l'AST di un'espressione e a restituire `true` se tale espressione è immutabile, `false` altrimenti. La politica applicata nel decretare l'immutabilità di un'espressione è la seguente:

- **Operatori binari**: sempre immutabili
- **Operatori unari (matematici)**: sempre immutabili
- **Operatori unari (logici)**: sempre immutabili
- **Operatori unari (su puntatori)**: immutabili se è immutabile l'operando
- **Cast**: immutabili se è immutabile l'operando
- **Accessi a membro**: immutabili se è immutabile l'operando
- **Accessi a cella**: immutabili se è immutabile l'operando
- **Chiamate a funzioni**: il valore restituito da una chiamata è *non* immutabile
- **Identificatori**: immutabili se dichiarati come costanti
- **Literal**: sempre immutabili

La classe `AssignmentImmutabilityChecker` verifica se è possibile l'espressione presa in easame ad un'espressione destinazione *non* immutabile. Tale classe è utilizzata dalla classe `ImmutabilityConstraintValidator` per ispezionare tutti gli assegnamenti.

La politica applicata nel decretare la validità di un assegnamento rispetto all'immutabilità è la seguente:

- L'assegnamento è *non-valido* se l'espressione destinazione è immutabile. Per valutare questo criterio ci si affida alla classe `ImmutabilityDeducer`
- L'assegnamento è *non-valido* se l'espressione destinazione è in sola lettura, ovvero se mutamenti di tale espressione non sarebbero osservabili in nessun caso. Per valutare questo criterio ci si affida alla classe `OvservabilityDeducer`
- L'assegnamento è *non-valido* se esso consentirebbe di modificare indirettamente l'espressione sorgente mediante manipolazione dell'espressione destinazione e l'espressione sorgente è immutabile. Per valutare questo criterio ci si affida alla classe `BondInspector`

Si ricordi che con *espressioni di sola-lettura* si intendono le espressioni che non sono propriamente immutabili, ma che qualora mutate non mostrerebbero effetti nel programma di alcun tipo. Un esempio di espressione immutabile è una chiamata a funzione che restituisce un non-puntatore (nè vettoriale nè scalare).

2.9 Backend: Utilizzo di LLVM per generare IR

In questa sezione sarà affrontato il tema di come è stato utilizzato il framework "LLVM" per generare codice IR, ottimizzarlo, e tradurlo in linguaggio macchina.

2.9.1 Traduzione dei tipi in LLVM-IR

La traduzione delle definizioni di tipo e delle type-signatures in LLVM-IR mediante è responsabilità della classe `TypesLLVMTranslator`. Questa classe è in grado di tradurre sia i tipi primitivi che i tipi definiti dall'utente.

```
class TypesLLVMTranslator {

    public:
        TypesLLVMTranslator(
            ProgramRepresentation& program_representation,
            llvm::LLVMContext& context,
            llvm::Module& llvm_module
        );

        llvm::Type* translate_type_definition(TypeDefinition&);
        llvm::Type* translate_typesignature(TypeSignature&);
        vector<llvm::Type*> translate_all(vector<TypeSignature>&);

        size_t compute_sizeof(const TypeSignature& typesignature);

        llvm::GlobalVariable* fetch_type_info(TypeSignature&);

    protected:
        /* other methods */

    private:
        ProgramRepresentation& program_representation;
        llvm::LLVMContext& context;
        llvm::Module& llvm_module;

        unordered_map<string, llvm::Type*>
            llvm_type_definitions;

        unordered_map<string, llvm::GlobalVariable*>
            llvm_exact_type_infos;

        unordered_map<string, std::vector<llvm::GlobalVariable*>>
            llvm_compatible_type_infos;
};
```

La traduzione di un tipo in LLVM-IR comporta la creazione di un oggetto di tipo `llvm::Type` che rappresenta una typesignature in LLVM-IR. In particolare, di tale oggetto si tende a mantenere solo un puntatore, in quanto l'API di LLVM lavora solo ed esclusivamente con puntatori. La *lifetime* di tali oggetti è gestita da LLVM stesso, ed in particolare essi sono automaticamente deallocati quando il modulo che li contiene viene deallocato. Come si può vedere, la classe `TypesLLVMTranslator` mantiene un riferimento a `llvm::Module` che corrisponde appunto al modulo appena citato.

La traduzione delle definizioni di struct avviene in modo molto naturale e diretto, in quanto LLVM-IR supporta le struct nativamente, ed è quindi sufficiente utilizzare l'API di LLVM opportunamente.

```
TypesLLVMTranslator.translate_struct_definition(struct_def):
    fully_qualified_name = program_representation
        .get_fully_qualified_typedefinition_name(struct_definition)

    if llvm_type_definitions[fully_qualified_name] != nil:
        return llvm_type_definitions[fully_qualified_name]

    llvm_type_def = llvm_create_struct(fully_qualified_name)
    llvm_type_definitions.insert(fully_qualified_name, llvm_type_def)
    for field : struct_definition.fields:
        llvm_type_def.fields.append(
            translate_typesignature(field.field_type)
        )

    if llvm_type_def.fields.is_empty():
        fields_types.append(llvm::Type::getInt1Ty(context))

    return llvm_type_def
```

La traduzione avviene solo la prima volta, tutte le altre volte ci si limita a recuperare il tipo già tradotto dalla mappa `llvm_type_definitions` usando il nome completamente qualificato come chiave di recupero.

Inserendo immediatamente il `llvm::Type*` nella mappa `llvm_type_definitions`, si fa sì che per definizioni ricorsive (indirette), si possa evitare di entrare in un loop infinito.

Per far si che sia possibile allocare oggetti il cui tipo è una struct senza field si è deciso di inserire un campo fittizio di un byte dentro ogni struct che non ha field. A seconda dell'architettura, tale campo potrebbe potrebbe portare ad una dimensione maggiore per motivi di alignment.

In generale, l'ordine di definizione di un field influenza l'alignment, così come avviene in C e C++. Sarebbe stato possible implementare un sistema di alignment personalizzato, ma ciò avrebbe compromesso l'interoperabilità con C.

Per la traduzione delle union invece, è stato necessario utilizzare un approccio differente, in quanto LLVM-IR non supporta le union nativamente. Per ovviare a questo problema, si è deciso di utilizzare una struct contenente due campi, corrispondneti ad header e payload. L'header è un puntatore che punta alla *typeinfo* dell'oggetto contenuto nel payload. Il payload è un array di byte di dimensione pari alla massima dimensione tra i campi della union. Entrambi questi campi sono definiti in base all'output dei metodi `fetch_type_info` e `compute_sizeof`.

```
TypesLLVMTranslator.translate_struct_definition(union_def):
    fully_qualified_name = program_representation
        .get_fully_qualified_typedefinition_name(union_def)

    if llvm_type_definitions[fully_qualified_name] != nil:
        return llvm_type_definitions[fully_qualified_name]

    llvm_type_def = llvm_create_struct(fully_qualified_name)
    llvm_type_definitions.insert(fully_qualified_name, llvm_type_def)
    for alternative : union_def.alternatives:
        union_payload_sizeof = max(
            compute_sizeof(alternative),
            union_payload_sizeof
        )

    llvm_type_def.fields = [
        llvm_create_pointer_to_type_info(),
        llvm_create_array_of_bytes(union_payload_sizeof)
    )

    return llvm_type_def
```

Per array e puntatori scalari la traduzione è banale in quanto l'API di LLVM prevede che dato un `llvm::Type*` si possa facilmente ottnere il `llvm::Type*` corrispondente ad un array di dimensione nota o un puntatore a tale tipo. Quando si incontra una `TypeSignature` si traduce in un `llvm::Type*` mediante traduzione diretta se è un tipo primitivo, oppure si recupera la definizione di tipo corrispondente e si restituisce la traduzione di tale definizione (il recupero restituisce una definizione già reificata).

Per i puntatori vettoriali si è deciso di utilizzare una traduzione basata su struct, in quanto LLVM-IR non supporta i vettori come tipi primitivi. In particolare, si è deciso di utilizzare una struct contenente un campo di tipo `uint64_t` che rappresenta la lunghezza del vettore, e un campo di tipo puntatore al tipo base del vettore.

2.9.2 Traduzione delle espressioni in LLVM-IR

Come già detto nella sezione di presentazione dell'LLVM-IR ad alto livello, le variabili in LLVM-IR sono solo simboli a cui sono associati dei valori, e non hanno un vero e proprio indirizzo di memoria. Per rappresentare una variabile in Basalt utilizzando LLVM-IR, è necessario avere due oggetti distinti, uno corrispondente al valore della variabile e uno corrispondente al suo indirizzo di memoria. Si utilizzerà dunque la seguente struct per modellare una generica espressione Basalt tradotta in LLVM-IR:

```
struct TranslatedExpression {
    llvm::Value* value;
    llvm::Value* address;
};
```

Si consideri l'espressione Basalt corrispondente al solo identifier `x`, e si assuma che tale identifier corrisponda ad una variabile intera con valore 6 allocata ad indirizzo di memoria `0xF5FAC43D`. La sua traduzione in una `TranslatedExpression` avrebbe la forma:

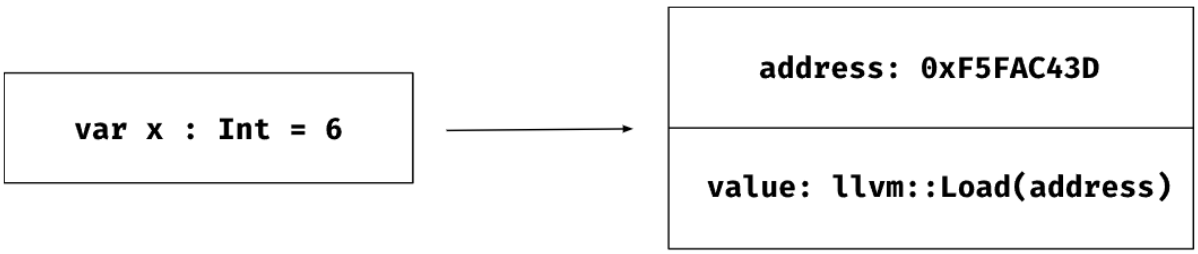


Figura 27: Traduzione di un identifier in LLVM-IR

Si noti che non ogni `TranslatedExpression` ha un indirizzo di memoria associato, ad esempio l'espressione `2` sarebbe tradotta come segue:

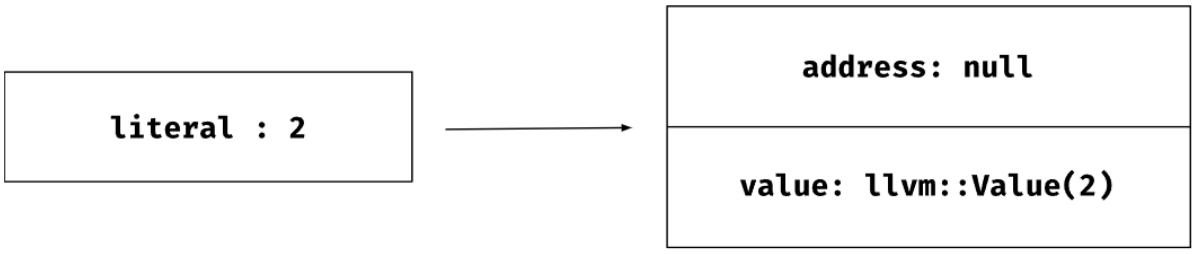


Figura 28: Traduzione di una literal in LLVM-IR

La traduzione dell'operatore unario di referenziazione (`&`) è un po' più complessa, in quanto è necessario tradurre l'operando in una `TranslatedExpression` e applicare ad essa una trasformazione per ottenere una nuova `TranslatedExpression` che rappresenti l'indirizzo di memoria dell'operando. Si consideri l'espressione `&x`, essa sarebbe dunque tradotta come segue:

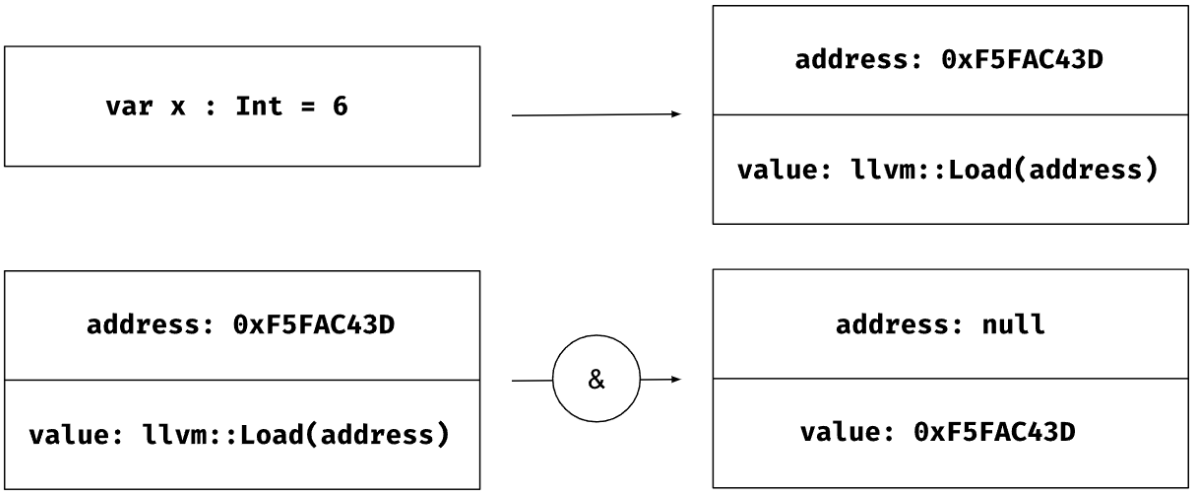


Figura 29: Traduzione di una referenziazione in LLVM-IR

Una dereferenziazione, così come la dereferenziazione (`#`), è un'operatore unario, e come tale, richiede di essere tradotto mediante la trasformazione della `TranslatedExpression` che rappresenta il suo operando. Si consideri l'espressione `#y`, dove y è un puntatore ad intero. Essa sarebbe tradotta come segue:

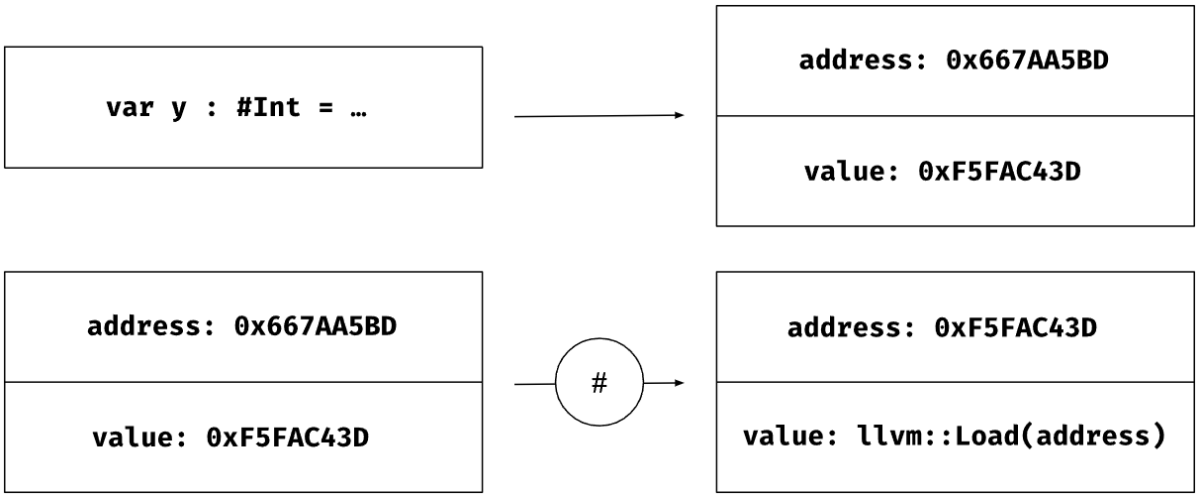


Figura 30: Traduzione di una dereferenziazione in LLVM-IR

Un generico operatore richiede la traduzione dei suoi operandi in `TranslatedExpression`, e la successiva applicazione di una qualche fuzione dell'API di LLVM ai valori di tutte le `TranslatedExpression` ottenute. Si consideri l'espressione `a + b`, essa sarebbe tradotta come segue:

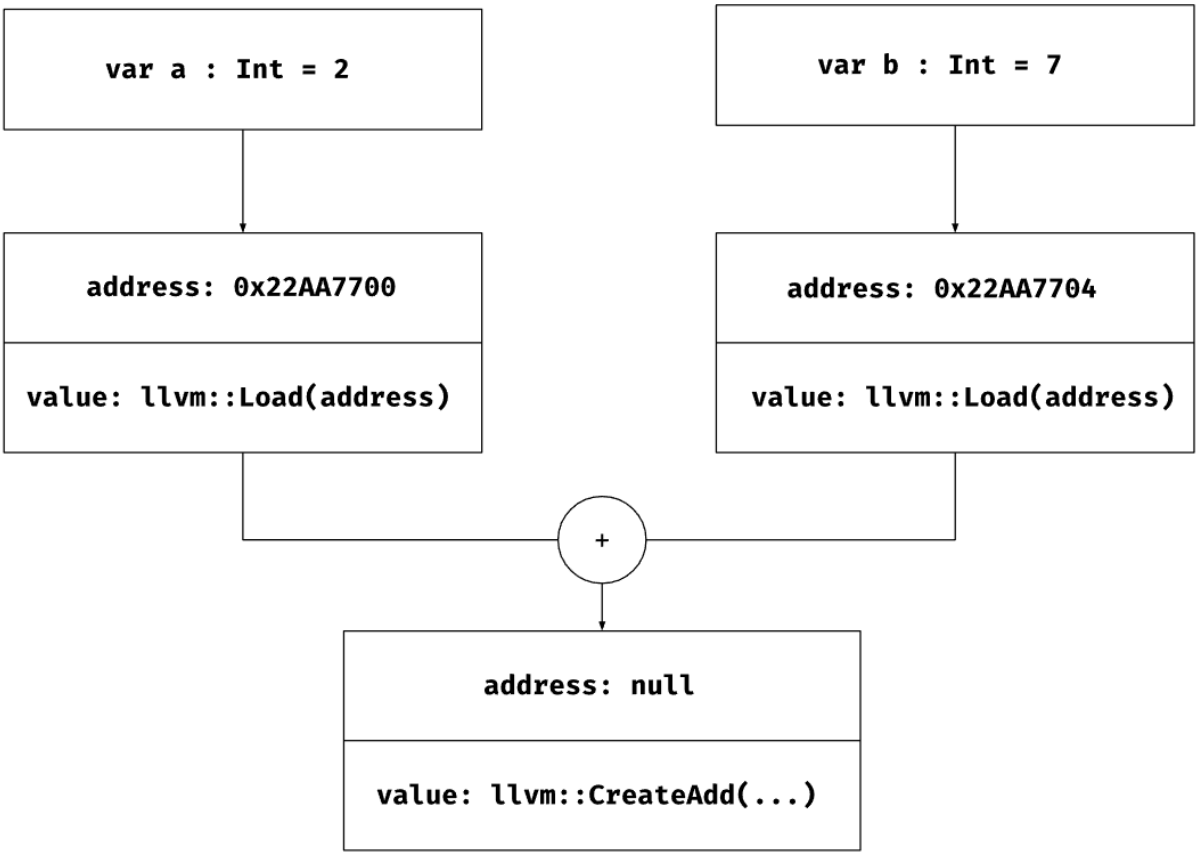


Figura 31: Traduzione di una somma in LLVM-IR

Lo stesso vale anche per operatori unari come l'operatore di negazione logica (`!`). Le espressioni di tipo chiamata a funzione, operatore `as` ed operatore `is` sono trattate in dettaglio nelle sezioni successive del documento.

Un'array literal viene tradotta come un'allocazione di memoria, seguita da una serie di assegnazioni ai singoli elementi dell'array. L'optimization engine di LLVM si occuperà di effettuare l'*inlining* delle assegnazioni, se possibile, o addirittura di rimuovere del tutto l'allocazione ed utilizzerà direttamente i valori la dove servono. Ovviamente, ciò è possibile solo se l'array literal viene usata in contesti dove i valori sono noti a priori e/o sono noti a priori gli indici ai quali saranno effettuate letture.

2.9.3 Traduzione degli statements in LLVM-IR

La traduzione degli statements in LLVM-IR è abbastanza diretta per quanto riguarda istruzioni di assegnamento, allocazioni, return e simili, ma è invece decisamente più complessa per quanto riguarda le istruzioni di controllo del flusso di esecuzione quali branch condizionali e cicli iterativi.

In generale, in LLVM, attraverso l'API in C++, si può inserire uno statement all'interno di un blocco di codice, a sua volta all'interno di una funzione, e si deve concludere ogni blocco con un istruzione di salto.

La traduzione di uno o più statement in LLVM-IR, così come la traduzione di una o più espressioni, è responsabilità della classe `ExpressionsAndStatementsLLVMTranslator`. Questa classe è stata progettata per essere istanziata una volta per ogni blocco di codice delimitato da parentesi graffe, ed è equipaggiata con tutti i metodi necessari a tradurre l'intero contenuto di tale blocco nonchè dei blocchi in esso innestati (essa è capace di generare autonomamente cloni di se stessa specializzati per i blocchi innestati che le si richiede di tradurre).

```
class ExpressionsAndStatementsLLVMTranslator {

    public:
        ExpressionsAndStatementsLLVMTranslator(
            /* other parameters */
            llvm::BasicBlock* loop_entry_block = nullptr,
            llvm::BasicBlock* loop_exit_block = nullptr
        );

        llvm::BasicBlock* translate_statement(llvm::Block*, ...);
        llvm::BasicBlock* translate_conditional(llvm::Block*, ...);
        llvm::BasicBlock* translate_while_loop(llvm::Block*, ...);
        llvm::BasicBlock* translate_until_loop(llvm::Block*, ...);

        TranslatedExpression translate_expression_to_llvm(...);
        TranslatedExpression translate_type_operator(...);

        /* other methods */

    protected:
        /* other methods */

    private:
        /* other fields */
        llvm::BasicBlock* loop_entry_block;
        llvm::BasicBlock* loop_exit_block;
};
```

La traduzione di un branch condizionale (if-statement) schematizzabile con come segue:

```
Translator.translate_conditional(current_block, conditional):
    if_cond_block = create_block_after(current_block)
    if_then_block = create_block_after(if_cond_block)
    if_else_block = create_block_after(if_then_block)
    if_exit_block = create_block_after(if_else_block)

    current_block.create_jump(if_cond_block)

    condition = translate_expression_to_llvm(
        if_cond_block,
        conditional.condition
    )
    if_cond_block.create_conditional_jump(
        condition.value,
        if_then_block,
        if_else_block
    )

    auto then_translator = create_translator_for_nested_conditional()
    if_then_block = then_translator.translate_all(
        if_then_block,
        conditional.then_branch
    )
    if_then_block.create_jump(if_exit_block)

    auto else_translator = create_translator_for_nested_conditional()
    if_else_block = else_translator.translate_all(
        if_else_block,
        conditional.else_branch
    )
    if_else_block.create_jump(if_exit_block)

    return if_exit_block
```

Ogni chiamata ai metodi che traducono in LLVM-IR uno statement restituisce il blocco di codice corrente al termine della traduzione di tale statement. Questo permette di concatenare le traduzioni di più statement in modo relativamente semplice. Al termine della traduzione di questo if-statement, il nuovo codice sarà scritto nel blocco chiamato "exit" che corrisponde al blocco a cui si effettua salto non condizionato al termine di entrambi i branch (sia "then" che "else"). La creazione di un blocco apposito per la condizione non era necessaria ma rende il codice IR generato più leggibile quando emesso in formato testuale, e ciò è utile a fini di debugging.

La traduzione di un ciclo while è schematizzabile come segue:

```
Translator.translate_while_loop(llvm_block, while_loop):
    while_cond_block = create_block_after(llvm_block)
    while_body_block = create_block_after(while_cond_block)
    while_exit_block = create_block_after(while_body_block)

    llvm_block.create_jump(while_cond_block);

    condition = translate_expression_to_llvm(
        while_cond_block,
        while_loop.condition
    )
    while_cond_block.create_conditional_jump(
        condition.vale,
        while_body_block,
        while_exit_block
    )

    body_translator = create_translator_for_nested_loop(
        while_cond_block,
        while_exit_block
    )
    while_body_block = body_translator.translate_all(
        while_body_block,
        while_loop.body
    )
    while_body_block.create_jump(while_cond_block)

    return while_exit_block
```

La chiamata a `create_translator_for_nested_loop` è responsabile della creazione di un nuovo oggetto `ExpressionsAndStatementsLLVMTranslator` che sarà utilizzato per tradurre il corpo del ciclo. Tale oggetto è configurato in modo da avere come blocco di entrata il blocco della condizione del ciclo e come blocco di uscita.

Ciò significa che in caso di istruzioni `break` o `continue` all'interno del ciclo, tali blocchi saranno utilizzati per effettuare rispettivamente o il salto al termine del ciclo o al termine dell'iterazione corrente.

La traduzione di un ciclo until (simile al do-while) è schematizzabile come segue:

```
Translator.translate_until_loop(llvm_block, until_loop):
    auto until_body_block = create_block_after(llvm_block)
    auto until_cond_block = create_block_after(until_body_block)
    auto until_exit_block = create_block_after(until_cond_block)

    current_block.create_jump(until_body_block)

    auto body_translator = create_translator_for_nested_loop(
        until_body_block,
        until_exit_block
    )
    until_body_block = body_translator.translate_all(
        until_body_block,
        until_loop.body
    )
    until_body_block.create_jump(until_cond_block)

    condition = translate_expression_to_llvm(
        until_cond_block,
        until_loop.condition
    )
    negated_condition = llvm_create_negation(condition.value)
    until_cond_block.create_conditional_jump(
        negated_condition.value,
        until_body_block,
        until_exit_block
    )

    return until_exit_block
```

È possibile schematizzare il metodo `translate_all` come segue:

```
Translator.translate_all(llvm_block, statements):
    for statement in statements:
        llvm_block = translate_statement(llvm_block, statement)
    return llvm_block
```

2.9.4 Conversioni implicite ed operatori di tipo

In sede di chiamata a funzione, di return da una funzione e di assegnamento è spesso necessario applicare conversioni implicite di tipo. Ad esempio, assegnando un `Int` ad un `Int|Float`, esso deve venire prima convertito, ovvero deve essere creata una union implicitamente a partire dal valore intero, e solo dopo tale union potrà essere assegnata.

Assegnando un puntatore ad array ad una slice, assegnando una slice ad una stringa e così via, è necessario applicare conversioni di tipo implicite, e ciò è responsabilità della classe `TypeManipulationsLLVMTranslator`.

```
class TypeManipulationsLLVMTranslator {

    public:
        TypeManipulationsLLVMTranslator(
            ProgramRepresentation& program_representation,
            TypesLLVMTranslator& types_llvm_translator
        );

        TranslatedExpression test_concrete_type_of_union(
            llvm::BasicBlock* block,
            TranslatedExpression union_expression,
            const TypeSignature& type_to_check
        );

        TranslatedExpression cast_translated_expression_as_copy(
            llvm::BasicBlock* block,
            TranslatedExpression expression,
            const TypeSignature& expression_original_type,
            const TypeSignature& dest_type
        );

        TranslatedExpression cast_translated_expression_as_ref(
            llvm::BasicBlock* block,
            TranslatedExpression expression,
            const TypeSignature& expression_original_type,
            const TypeSignature& dest_type
        );

    private:
        /* other fields */
        /* other methods */
};
```

Nei casi di conversione implicita in sede di return, di assegnamento o di un parametro attuale al tipo di un parametro formale di una funzione in sede di chiamata la conversione porta ad una copia, mentre l'utilizzo dell'operatore `as` converte per riferimento.

In entrambi i casi, l'approccio scelto è stato quello di generare una `CastStrategy` basandosi sul tipo attuale dell'espressione da convertire e sul tipo di destinazione. Questa strategia è poi utilizzata per effettuare un dispatch e chiamare un metodo opportuno per la gestione del caso concreto.

```
class TypeManipulationsLLVMTranslator {

    public:
        /* other methods */

        CastStrategy compute_cast_strategy(
            TypeSignature,
            TypeSignature
        );

    private:
        enum class CastStrategy {
            noop,
            union_to_union,
            union_to_alternative,
            alternative_to_union,
            array_pointer_to_slice,
            array_pointer_to_string,
            array_pointer_to_raw_string,
            slice_to_string,
            slice_to_raw_string,
            string_to_raw_string,
            array_to_array,
        };

        /* other fields */
        /* other methods */
};
```

I due metodi pubblici di cui sopra `cast_translated_expression_as_copy` e `cast_translated_expression_as_ref` si occupano di effettuare un dispatch basandosi sulla strategia di cast calcolata, e di chiamare il metodo (privato) opportuno per la gestione del caso concreto tenendo conto del fatto che serve effettuare una conversione rispettivamente o per copia o per riferimento.

Il metodo `test_concrete_type_of_union` si occupa di implementare sostanzialmente l'operatore `is`, esso accetta in input una espressione di tipo union, estrae il suo header, che si ricorda essere un puntatore ad una globale che traccia il tipo concreto contenuto nel payload, e lo paragona con la variabile globale restituita dal metodo `fetch_type_info`, della classe `TypesLLVMTranslator`, chiamata sulla `TypeSignature` da investigare.

Anche se le due variabili globali non coincidessero servirà comunque vedere se la `TypeSignature` da investigare non sia una union. In caso essa sia una union, bisognerà investigare ricorsivamente tutte le alternative che essa codifica.

L'operatore `is` restituirà `true` se il tipo concreto tracciato dall'header è uguale al tipo che si sta investigando o ad una qualunque delle sue alternative. Ciò perchè arrivati alla fase di generazione dell'IR, il typechecking è già avvenuto e ci si è già assicurati del fatto che la `TypeSignature` usata come operando destro sia un sottotipo del tipo dichiarato dell'operando sinistro.

```
Translator.test_concrete_type_of_union(block, union, type):
    union_header_address = block.create_struct_access(union, 0)
    union_header = block.create_load(union_header_address)

    expected_type_info = type_definitions_llvm_translator
        .fetch_type_info(type)
    casted_expected_type_info = block
        .create_bitcast(expected_type_info, union_header.type)

    is_operator_result = builder
        .create_compare(union_header, casted_expected_type_info)

    alternative_type_infos = type_definitions_llvm_translator
        .fetch_all_type_infos_for_non_union_compatible_types(type)

    for alternative_type_info in alternative_type_infos:
        casted_alternative_type_info = block.create_bitcast(
            alternative_type_info,
            union_header.type
        )
        is_current_type = builder.create_compare(
            union_header,
            casted_alternative_type_info
        )
        is_operator_result = block.create_or(
            is_operator_result,
            is_current_type
        )

    return is_operator_result;
```

2.9.5 Traduzione delle funzioni in LLVM

La traduzione delle funzioni in codice LLVM è responsabilità della classe `CallableCodeBlocksLLVMTranslator`, la quale si occupa di tradurre un generico `CallableCodeBlock`, sia che esso abbia tipo concreto `FunctionDefinition` sia che esso abbia tipo concreto `CFAPlanDescriptor`.

```
class CallableCodeBlocksLLVMTranslator {

    public:
        CallableCodeBlocksLLVMTranslator(
            ProgramRepresentation& program_representation,
            TypeDefinitionsLLVMTranslator& types_translator,
            llvm::LLVMContext& llvm_context,
            llvm::Module& llvm_module
        );

        llvm::Function* translate_callable_code_block_to_llvm(
            const CallableCodeBlock& callable_code_block
        );

        llvm::Function* translate_cfa_descriptor_to_llvm(
            const CFAPlanDescriptor& cfa_plan_descriptor,
            llvm::Function* llvm_function
        );

        llvm::Function* translate_function_definition_to_llvm(
            const FunctionDefinition::Ref& function_definition,
            llvm::Function* llvm_function
        );

    protected:
        /* other methods */

    private:
        /* other fields */
};
```

Il metodo `translate_callable_code_block_to_llvm` si occupa di effettuare un dispatch sui due metodi `translate_cfa_descriptor_to_llvm` e `translate_function_definition_to_llvm` in base al tipo concreto dell'oggetto passato come parametro.

Il metodo `translate_function_definition_to_llvm` si occupa di tradurre una definizione di funzione in codice LLVM. Questo metodo è responsabile di creare un nuovo `llvm::Function` all'interno del modulo LLVM, e di tradurre il corpo della funzione in codice LLVM.

Una funzione non extern sarà tradotta con un corpo composto da almeno un blocco, chiamato *entry*, in accordo con le specifiche di LLVM. In tale blocco saranno presenti le allocazioni su stack necessarie per poter avere oggetti di tipo `TranslatedExpression` che rappresentino i parametri formali della funzione. Senza soluzioni di continuità tale blocco sarà utilizzato come blocco iniziale in cui chiamare il metodo `translate_all` della classe `ExpressionsAndStatementsLLVMTranslator`.

```
Translator.translate_local_function_definition_to_llvm(
    func_def, llvm_function
):
    entry_block = llvm_create_block("entry", llvm_function)
    body_translator = ExpressionsAndStatementsLLVMTranslator(
        func_def, entry_block, ...
    )
    body_translator.alloc_args(func_def, entry_block)
    exit_block = body_translator.translate_all(
        func_def, entry_block, ...
    )
    inject_return_statement_if_needed(exit_block, ...)
    return llvm_function
```

Se la funzione è stata dichiarata come extern, allora la traduzione sarà effettuata sotto forma di dichiarazione di simbolo esterno, primitiva dell'IR di LLVM.

```
Translator.translate_extern_function_definition_to_llvm(
    func_def, llvm_function
):
    llvm_function.delete_body()
    llvm_function.set_linkage(llvm.ExternalLinkage)
    llvm_function.set_name(func_def.extern_name)
    return llvm_function
```

Una funzione senza corpo, e con policy di linkage "external", sarà tradotta come una dichiarazione di simbolo esterno. Ad esempio, la dichiarazione della funzione `square_root(Float) -> Float` in Basalt, viene tradotta in LLVM-IR come segue:

```
//BASALT
extern square_root(x: Float) -> Float = "sqrt";
```

```
;;LLVM-IR
declare double @sqrt(double)
```

2.9.6 Traduzione degli overload CFA in LLVM

La traduzione in LLVM-IR degli overload CFA è responsabilità della classe `CallableCodeBlocksLLVMTranslator` così come già detto in precedenza. La traduzione inizia dalla creazione di un *entry-block* all'interno della funzione LLVM-IR che si sta creando, per poi proseguire ricorsivamente traducendo all'interno del blocco corrente (che all'inizio è proprio l'entry-block) l'albero decisionale codificato dal `CFAPlanDescriptor`.

```
Translator.translate_cfa_descriptor_to_llvm(
    CFAPlanDescriptor descriptor,
    llvm_function
):
    entry_block = llvm_create_block("entry", llvm_function)
    translate_cfa_plan_to_llvm(
        descriptor,
        descriptor.plan,
        llvm_function,
        entry_block
    )
    return llvm_function
```

```
Translator.translate_cfa_plan_to_llvm(
    CFAPlanDescriptor cfa_plan_descriptor,
    CFAPlan cfa_plan,
    llvm_function,
    current_block
):
    if (cfa_plan.is_direct_adoption()):
        selected_concrete_function = cfa_plan.get_direct_adoption()
        translate_cfa_direct_adoption_to_llvm(
            cfa_plan_descriptor,
            selected_concrete_function,
            llvm_function,
            current_block
        )
    else:
        recursive_plan = cfa_plan.get_recursive_adoption()
        translate_cfa_recursive_adoption_to_llvm(
            cfa_plan_descriptor,
            recursive_plan,
            llvm_function,
            current_block
        )
```

La traduzione di una `FunctionDefinition` sotto forma di caso base dell'albero decisionale non è in alcun modo difforme da una normale chiamata a funzione effettuata all'interno del blocco corrente.

La traduzione invece di un `RecursiveCFAPlan` è più complessa, in quanto richiede la creazione di un nuovo blocco per ogni figlio del nodo dell'albero decisionale che esso codifica.

```
Translator.translate_cfa_recursive_adoption_to_llvm(
    CFAPlanDescriptor cfa_plan_descriptor,
    RecursiveAdoptionPlan recursive_plan,
    llvm_function,
    current_block
):
    alternative_blocks = { current_block };
    for alt_index = 1 ... recursive_plan.alternatives.size():
        prev_block = alternative_blocks.[alt_index - 1]
        alternative_block = llvm_create_block("", llvm_function)
        alternative_blocks.push_back(alternative_block)

    for alt_index = 0 ... recursive_plan.alternatives.size() - 1:
        alternative_block = alternative_blocks[alt_index]
        llvm_argument = llvm_function.args[recursive_plan.arg_index]
        helper = TypeManipulationsLLVMTranslator(...)
        is_operator = helper.test_concrete_type_of_union_in_llvm(
            alternative_block,
            llvm_argument,
            recursive_plan.alternatives[alt_index]
        )

        success_block = llvm_create_block("", llvm_function)
        alternative_block_builder.create_conditional_jump(
            is_operator.value,
            success_block,
            alternative_blocks[alt_index + 1]
        )

        successful_plan = recursive_plan.nested_plans[alt_index];
        translate_cfa_plan_to_llvm(
            cfa_plan_descriptor,
            successful_plan,
            llvm_function,
            success_block
        )

    // handle last alternative as successful
    translate_cfa_plan_to_llvm(...)
```

In totale, il numero di blocchi di una funzione in LLVM-IR che implementa un overload autogenerato tramite CFA avrà un numero di blocchi calcolabile come:

$$\text{numero di blocchi} = \sum_{i=1}^{U}(2K_i - 1)$$

- U è il numero di argomenti il cui tipo è una union
- K_i è il numero di sottotipi diretti della i-esima union tra gli argomenti

Si consideri ad esempio il caso di una ipotetica chiamata alla funzione `double` effettuata con un solo argomento di tipo `Int|Float`, dunque da risolvere mediante CFA date le due definizioni, le quali coprono tutti i casi:

```
func double(x : Int) -> Int { return x * 2; }
func double(x : Float) -> Float { return x * 2.0; }
```

Il numero di blocchi necessari per tradurre l'overload autogenerato in LLVM-IR è 3, in accordo con la forma generale della formula sopra esposta nel caso in cui $U = 1$ e $K_1 = 2$.

L'overload sarà tradotto in LLVM-IR come segue, al netto delle opportune semplificazioni:

```
define void @cfa_double(%"Int|Float" %0) {
    entry:
        ...
        ;; check if the argument is an Int
        %flag = icmp eq i8* %3, getelementptr inbounds (...)
        br i1 %flag, label %IntCase, label %FloatCase

    IntCase:
        ...
        %out = call i64 int_double(i64 %9)
        ...
    FloatCase:
        ...
        %out = call f64 float_double(f64 %14)
        ...
    }
```