

Analisi dei Sistemi Informatici

Dominio Astratto degli Intervalli

Autori:

Marco Colognese VR423791

Mattia Rossini VR423614

Indice

Introduzione	2
Concetti di base	3
Analisi Statica	4
Interpretazione Astratta	4
Dominio astratto degli Intervalli	6
Semantica astratta delle espressioni	6
Semantica astratta dei comandi	7
Accelerazione della convergenza	7
Widening	7
Narrowing	7
Implementazione	8
Libreria	8
Linguaggio <i>Toy</i>	10
Interprete per il linguaggio	11
Test	12
Documentazione	12
Conclusioni	13

Introduzione

Lo scopo principale di questo progetto è quello di fornire un'implementazione di una libreria basata sul *dominio astratto degli intervalli*, scritta nel linguaggio *C++*. Essa permette di effettuare l'analisi statica di un qualsiasi programma, sfruttando le proprietà e le operazioni di tale dominio.

La motivazione principale che ci ha portato allo sviluppo del seguente progetto è l'interesse per l'approfondimento dell'argomento trattato a lezione. Inoltre abbiamo ritenuto utile implementare questa libreria per vedere l'applicazione effettiva del dominio ad un programma reale.

Abbiamo colto anche l'occasione per implementare il tutto in *C++*, linguaggio di programmazione che stiamo da poco imparando attraverso il corso seguito durante il secondo semestre, che prevede la consegna di un progetto a scelta dello studente (collegato al proprio corso di studio) scritto in questo linguaggio.

L'argomento su cui si basa questo progetto è il dominio astratto degli intervalli per l'interpretazione astratta. Il contesto in cui ci si trova è il framework dell'interpretazione astratta, una delle tecniche principali usate per fare l'analisi statica di un programma.

Si basa sull'idea che è possibile tener traccia di tutte le possibili esecuzioni di un programma a tempo di compilazione: ciò è reso possibile applicando un certo grado di approssimazione. Il livello di approssimazione (e di conseguenza la precisione e l'efficienza dell'analisi) dipende dal dominio astratto che si utilizza per analizzare il programma. Infatti, dato un elemento concreto, ci sono varie possibili approssimazioni di esso: ciò significa che si possono utilizzare più domini astratti per approssimare lo stesso elemento. Ogni dominio andrà a preservare differenti tipi di informazione e sono utili per dimostrare diversi tipi di proprietà.

Il progetto sviluppato comprende le seguenti sezioni:

- implementazione di una libreria suddivisa in classi per poter effettuare l'analisi di un programma attraverso gli intervalli corrispondenti a ciascuna variabile;
- implementazione di un interprete per un semplice linguaggio di programmazione *Toy*, di cui abbiamo prodotto sintassi e semantica;
- implementazione di test per ciascuna classe e funzione in essa contenuta, al fine di ottenere una buona copertura del codice;
- generazione automatica della documentazione di tutto il progetto, partendo dal codice sorgente, attraverso il tool *Doxygen*.

Concetti di base

Definizione 1 Un poset $\langle X, \leq \rangle$ è un insieme dotato di ordine parziale \leq su X (relazione binaria riflessiva, antisimmetrica e transitiva).

Definizione 2 Un poset $\langle P, \leq \rangle$ soddisfa la condizione di catena ascendente (ACC) se e solo se ogni sequenza infinita $x_0 \leq x_1 \leq \dots x_n \leq \dots$ di un elemento x_n di P non è strettamente crescente, cioè $\exists k \geq 0 : \forall j \geq k : x_k = x_j$.

Definizione 3 Un poset $\langle P, \leq \rangle$ soddisfa la condizione di catena decrescente (DCC) se e solo se ogni sequenza infinita $x_0 \geq x_1 \geq \dots x_n \geq \dots$ di un elemento x_n di P non è strettamente decrescente, cioè $\exists k \geq 0 : \forall j \geq k : x_k = x_j$.

Definizione 4 Un poset $\langle P, \leq \rangle$ ha:

- un elemento top \top se e solo se $\top \in P \wedge \forall x \in P : x \leq \top$;
- un elemento bottom \perp se e solo se $\perp \in P \wedge \forall x \in P : \perp \leq x$.

Se gli elementi top e bottom esistono allora questi sono unici.

Definizione 5 Sia $\langle P, \leq \rangle$ un poset, allora:

- $M \in P$ è un upper bound di $S \subseteq P$ se e solo se $\forall x \in S : x \leq M$;
- $m \in P$ è un lower bound di $S \subseteq P$ se e solo se $\forall x \in S : m \leq x$.

Definizione 6 Sia $\langle P, \leq \rangle$ un poset e $X \subseteq P$. Il least upper bound \sqcup di X , se esiste, è x tale che:

- x è un upper bound di X ;
- $\forall u \in P : (\forall y \in X : y \leq u) \Rightarrow (x \leq u)$.

Dualmente è possibile definire il greatest lower bound \sqcap di X .

Definizione 7 Un semireticolo join $\langle P, \leq, \sqcup \rangle$ è un poset $\langle P, \leq \rangle$ tale che due elementi $x, y \in P$ hanno un least upper bound $x \sqcup y$.

Definizione 8 Un semireticolo meet $\langle P, \leq, \sqcap \rangle$ è un poset $\langle P, \leq \rangle$ tale che due elementi $x, y \in P$ hanno un greatest lower bound $x \sqcap y$.

Definizione 9 Un reticolo $\langle P, \leq, \sqcup, \sqcap \rangle$ è sia un semireticolo join $\langle P, \leq, \sqcup \rangle$ che un semireticolo meet $\langle P, \leq, \sqcap \rangle$.

Definizione 10 Un reticolo completo è un poset $\langle P, \sqsubseteq \rangle$ tale che ogni sottoinsieme $X \subseteq P$ ha un least upper bound $\sqcup X$ in P .

Definizione 11 Un punto fisso di un operatore f su un insieme L è $x \in L$ tale che:

$$f(x) = x$$

Un operatore può avere zero, uno oppure più punti fissi.

Analisi Statica

I programmi informatici spesso contengono errori ed in particolare:

- eseguire un programma contenente alcuni *bug* può portare a conseguenze indesiderate, più o meno gravi;
- testare il codice non è sufficiente, poiché è possibile ricoprire solo una parte delle possibili tracce di esecuzione.

Dunque quando sono richieste garanzie importanti riguardanti la correttezza di un certo programma, è necessario fare uso dell'analisi statica.

L'*analisi statica* del codice di un programma ci permette, tramite una serie di tecniche, di calcolare un'approssimazione sicura dell'insieme dei valori o dei comportamenti che si verificheranno durante l'esecuzione di tale programma.

Lo scopo principale consiste nell'evitare ai compilatori inutili calcoli ridondanti. Un esempio può essere l'eliminazione di *dead code* che permette di togliere delle parti di codice che il programma, durante la sua esecuzione, non raggiungerà mai.

L'idea che sta sotto all'analisi statica consiste nel ricavare qualche informazione del programma senza doverlo eseguire; per il *Teorema di Rice* ciò è indecidibile.

Teorema 1 (Teorema di Rice) *Ogni proprietà non banale dei linguaggi ricorsivamente enumerabili è indecidibile.*

Dunque occorre introdurre un certo grado di approssimazione. Se si denota con $\llbracket \cdot \rrbracket$ il comportamento concreto del programma e con $\llbracket \cdot \rrbracket^\#$ il comportamento approssimato, si ottiene che:

$$\llbracket P \rrbracket \subseteq \llbracket P \rrbracket^\#$$

Uno dei principali approcci coinvolti nell'analisi statica è l'interpretazione astratta: essa modella il calcolo in termini di oggetto astratto per fornire alcune informazioni sui calcoli concreti.

Interpretazione Astratta

L'*interpretazione astratta* è stata sviluppata alla fine degli anni Settanta da Patrick Cousot e Radhia Cousot ed è un *framework* per l'analisi statica che definisce un'astrazione corretta per la semantica concreta di un programma (potenzialmente infinita).

Lo scopo principale è quello di approssimare le proprietà di interesse di un programma, ottenute mediante l'analisi statica, a delle strutture matematiche al fine di ottenere un'approssimazione di alcuni problemi complessi e critici dell'informatica, come la verifica, la validazione e la sicurezza dei sistemi informatici.

La semantica astratta necessita di un dominio astratto per "catturare" le proprietà di interesse. Si tratta, in questo caso, di un dominio astratto numerico non relazionale poiché astrae ogni variabile in modo indipendente dalle altre.

I domini numerici astratti sono importanti per gli analizzatori statici e vengono usati per verificare proprietà critiche di un programma come l'assenza di *buffer overflow* o eventuali divisioni per zero.

Per creare un'astrazione secondo l'idea di base dell'*interpretazione astratta* abbiamo bisogno di:

- un *dominio concreto* C che sarà il punto di partenza insieme alla *semantica concreta*. Esso definisce una descrizione formale di uno stato di calcolo, mentre la semantica concreta è una funzione definita sul dominio concreto e associa un significato alle dichiarazioni del programma. La semantica concreta è l'espressione matematica più precisa riguardo il comportamento del programma;
- un *dominio astratto* A che sarà un'approssimazione del dominio concreto. Modella alcune proprietà dei calcoli concreti, tralasciando le informazioni superflue;
- una *funzione di astrazione* $\alpha : C \rightarrow A$ che associa un elemento del dominio concreto C al suo corrispondente in quello astratto A ;
- una *funzione di concretizzazione* $\gamma : A \rightarrow C$ associa un elemento del dominio astratto A al suo corrispondente in quello concreto C .

Un aspetto importante del dominio astratto A è che un'astrazione $\alpha(x) \in A$ (con $x \in C$) è un'approssimazione di un valore nel dominio astratto e non la sua rappresentazione esatta:

$$X \subseteq \gamma(\alpha(X))$$

Definizione 12 Siano $\langle C, \leq_C \rangle$ e $\langle A, \leq_A \rangle$ due posets, $\alpha : C \rightarrow A$ e $\gamma : A \rightarrow C$. Diciamo che (C, α, γ, A) è una *connessione di Galois* se:

- α e γ sono monotone;
- $\forall c \in C : c \leq_C \gamma(\alpha(c))$;
- $\forall a \in A : \alpha(\gamma(a)) \leq_A a$.

Una connessione di Galois si indica con la seguente notazione:

$$\langle C, \leq_C \rangle \xleftrightarrow[\alpha]{\gamma} \langle A, \leq_A \rangle$$

Dominio astratto degli Intervalli

Il dominio astratto degli Intervalli è un dominio numerico non relazionale. Esso si basa sul concetto classico di *intervallo aritmetico*, introdotto negli anni '60 da *R. E. Moore*. È stato molto utilizzato nell'informatica ed è stato conseguentemente adattato per l'interpretazione astratta da *P. & R. Cousot*.

In questo dominio, un insieme di interi viene approssimato dal più piccolo intervallo che li contiene tutti, ad esempio l'insieme \mathbb{Z} viene approssimato con $[l, u]$, dove $l = \min \mathbb{Z}_{i \in \mathbb{N}}$, $u = \max \mathbb{Z}_{i \in \mathbb{N}}$. Tuttavia, non è sempre possibile conoscere il limite superiore o inferiore dell'insieme di interi, ciò ammette intervalli $[l, u]$ in cui l può essere $-\infty$ e/o u può essere $+\infty$.

$$\mathbb{I} = \{[l, u] \mid l \in \mathbb{Z} \cup \{-\infty\}, u \in \mathbb{Z} \cup \{+\infty\}, l \leq u\}$$

Questo dominio è infatti un reticolo completo. Si introduce l'ordinamento \sqsubseteq tale che $[a, b] \sqsubseteq [c, d]$ se e solo se l'intero intervallo $[a, b]$ è interamente contenuto in $[c, d]$, cioè $a \geq c$ e $b \leq d$. L'elemento *top* \top è l'intervallo $[-\infty, \infty]$, poiché contiene qualunque altro intervallo. L'elemento *bottom* \perp è l'insieme vuoto che non contiene alcun elemento. Questo reticolo ha altezza infinita e per questo motivo necessita di un operatore di *widening*.

Questo dominio è veloce e facile da utilizzare, ma ha alcuni difetti. Per esempio, considerando l'insieme $\{-10, 5\}$ (cardinalità = 2). Questo coppia di valori viene approssimata con l'intervallo $[-10, 5]$ (cardinalità = 16) che è un'approssimazione eccessivamente ampia.

Semantica astratta delle espressioni

- $[l_1, u_1] +^\# [l_2, u_2] = [l_1 + l_2, u_1 + u_2]$
- $-^\# [l, u] = [-u, -l]$
- $[l_1, u_1] *^\# [l_2, u_2] = [a, b]$ dove:
 - $a = \min(l_1 * l_2, l_1 * u_2, u_1 * l_2, u_1 * u_2)$
 - $b = \max(l_1 * l_2, l_1 * u_2, u_1 * l_2, u_1 * u_2)$
- $[l_1, u_1] =^\# [l_2, u_2] = \begin{cases} [1, 1] & \text{se } l_1 = l_2 = u_1 = u_2 \\ [0, 0] & \text{se } u_1 < l_2 \vee u_2 < l_1 \\ [0, 1] & \text{altrimenti} \end{cases}$
- $[l_1, u_1] <^\# [l_2, u_2] = \begin{cases} [1, 1] & \text{se } u_1 < l_2 \\ [0, 0] & \text{se } u_2 \leq l_1 \\ [0, 1] & \text{altrimenti} \end{cases}$

Semantica astratta dei comandi

$$D = (\mathbb{V}ar \rightarrow \mathbb{I}) \cup \{\perp\}$$

$$\begin{aligned} \llbracket ; \rrbracket^\# D &= D \\ \llbracket NonZero(e) \rrbracket^\# D &= \begin{cases} \perp & \text{se } [0, 0] = \llbracket e \rrbracket^\# D \\ D & \text{se } [0, 0] \neq \llbracket e \rrbracket^\# D \end{cases} \\ \llbracket Zero(e) \rrbracket^\# D &= \begin{cases} D & \text{se } [0, 0] \subseteq \llbracket e \rrbracket^\# D \\ \perp & \text{se } [0, 0] \not\subseteq \llbracket e \rrbracket^\# D \end{cases} \\ \llbracket x \leftarrow e \rrbracket^\# D &= D[x \mapsto \llbracket e \rrbracket^\# D] \\ \llbracket x \leftarrow M[e] \rrbracket^\# D &= D[x \mapsto [-\infty, +\infty]] \\ \llbracket M[e_1] \leftarrow e_2 \rrbracket^\# D &= D \end{aligned}$$

Accelerazione della convergenza

Il dominio degli Intervalli non rispetta la *ACC* (*ascending chain condition*), dunque non garantisce la terminazione: per questo viene introdotto il *widening*.

Widening

Un widening $[\nabla : P \times P \rightarrow P]$ su un poset $\langle P, \leq_P \rangle$ è una funzione che soddisfa:

- $\forall x, y \in P : x \leq (x \nabla y) \wedge y \leq (x \nabla y)$
- per ogni catena ascendente $x_0 \leq x_1 \leq \dots \leq x_n$ la catena definita come $y_0 = x_0, \dots, y_{n+1} = y_n \nabla x_{n+1}$ non è strettamente crescente.

Dato che in interpretazione astratta è necessario garantire/accelerare la convergenza, viene usato il widening (che si sostituisce al least upper bound), dal momento che anche il calcolo astratto può divergere. Il risultato di un widening è un post-punto fisso di F^∇ , ovvero una sovra-approssimazione del punto fisso più piccolo di f .

Il widening sul dominio degli intervalli funziona come segue:

$$[a, b] \nabla [c, d] = [e, f] \quad \text{t.c.} \quad e = \begin{cases} -\infty & \text{se } c < a \\ a & \text{altrimenti} \end{cases} \quad f = \begin{cases} +\infty & \text{se } b < d \\ b & \text{altrimenti} \end{cases}$$

Narrowing

Dato che il widening raggiunge un post-fixpoint, può capitare che si abbiano eccessive perdite di informazione, in questo caso viene usato il narrowing.

Il narrowing è una funzione $\triangle : P \times P \rightarrow P$ tale che:

- $\forall x, y \in P : y \leq x \implies y \leq x \triangle y \leq x$
- Per ogni catena discendente $x_0 \geq x_1 \geq \dots$, la catena discendente $y_0 = x_0, \dots, y_{i+1} = y_i \triangle x_{i+1}$ non è strettamente decrescente.

Per il dominio degli intervalli, il narrowing funziona come segue:

$$[a, b] \triangle [c, d] = [e, f] \quad \text{t.c.} \quad e = \begin{cases} c & \text{se } a = -\infty \\ a & \text{altrimenti} \end{cases} \quad f = \begin{cases} d & \text{se } b = +\infty \\ b & \text{altrimenti} \end{cases}$$

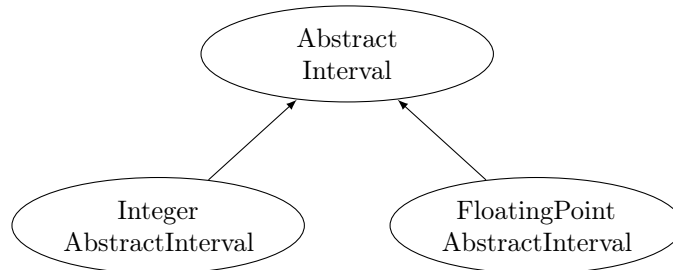
Implementazione

Libreria

La libreria implementata è composta dalle seguenti classi, contenenti i relativi metodi:

- *AbstractInterval*: classe astratta che rappresenta un intervallo generico con numeri interi oppure a virgola mobile. È composta dai seguenti metodi:
 - `getLowerBound()`: ritorna il lower bound di un intervallo generico;
 - `setLowerBound(Bound object)`: permette di settare il lower bound di un intervallo generico;
 - `getUpperBound()`: ritorna l'upper bound di un intervallo generico;
 - `setUpperBound(Bound object)`: permette di settare l'upper bound di un intervallo generico;
 - `isFinite()`: ritorna *true* se l'intervallo non contiene bound infiniti, *false* altrimenti;
 - `isContainedIn(AbstractInterval &i)`: ritorna *true* se l'intervallo generico è contenuto in quello ricevuto in input, *false* altrimenti;
 - `width()`: ritorna la cardinalità dell'intervallo;
 - `intersects(AbstractInterval &i)`: ritorna *true* se l'intervallo generico interseca quello ricevuto in input, *false* altrimenti.
- *IntegerAbstractInterval*: classe che estende *AbstractInterval* per intervalli i cui *bound* sono numeri interi. I metodi implementati (oltre a quelli ereditati) sono:
 - override di `operator+`, `operator-`, `operator*`, `operator/`, `operator==`: effettuano le operazioni ed i confronti tra due intervalli interi, gestendo tutti i casi particolari che si possono verificare;
 - `lub(IntegerAbstractInterval &i, IntegerAbstractInterval &ii)`: calcola il *least upper bound (lub)* tra due intervalli interi e lo ritorna;
 - `negate()` ritorna il negato dell'intervallo intero sul quale viene chiamato il metodo;
 - `widening(IntegerAbstractInterval &i)`: calcola il widening tra l'intervallo intero sul quale viene chiamato il metodo e quello ricevuto come parametro e lo ritorna;
 - `narrowing(IntegerAbstractInterval &i)`: calcola il narrowing tra l'intervallo intero sul quale viene chiamato il metodo e quello ricevuto come parametro e lo ritorna.
- *FloatingPointAbstractInterval*: classe che estende *AbstractInterval* per intervalli i cui *bound* sono numeri a virgola mobile. I metodi implementati (oltre a quelli ereditati) sono:
 - override di `operator+`, `operator-`, `operator*`, `operator/`, `operator==`: effettuano le operazioni ed i confronti tra due intervalli decimali, gestendo tutti i casi particolari che si possono verificare;
 - `lub(IntegerAbstractInterval &i, IntegerAbstractInterval &ii)`: calcola il *least upper bound (lub)* tra due intervalli decimali e lo ritorna;
 - `negate()` ritorna il negato dell'intervallo decimale sul quale viene chiamato il metodo;

- `widening(IntegerAbstractInterval &i)`: calcola il widening tra l'intervallo decimale sul quale viene chiamato il metodo e quello ricevuto come parametro e lo ritorna;
- `narrowing(IntegerAbstractInterval &i)`: calcola il narrowing tra l'intervallo decimali sul quale viene chiamato il metodo e quello ricevuto come parametro e lo ritorna.



- *Bound*: classe che rappresenta uno dei due *bound* che compongono un intervallo. Può essere generato partendo da un numero intero, a virgola mobile oppure infinito. I metodi che lo compongono sono i seguenti:

- `getIntValue()`: ritorna il valore del bound intero;
- `getInfinityValue()` ritorna il valore del bound infinito, con segno;
- `getFloatValue()` ritorna il valore del bound decimale;
- `getValue()` ritorna il valore del bound utilizzando il template per ritornare il tipo corretto;
- `is_infinity()` ritorna *true* se il bound è infinito;
- `is_integer()` ritorna *true* se il bound è intero;
- `is_float()` ritorna *true* se il bound è decimale;
- `revertSign()` ritorna il bound con il segno del valore invertito;
- override di `operator+`, `operator-`, `operator*`, `operator/`: effettuano le operazioni tra due bound, siano essi interi, decimali o infiniti, gestendo tutti i casi particolari che si possono verificare;
- override di `operator<`, `operator<=`, `operator>`, `operator>=`, `operator==`, `operator!=` gestiscono i confronti tra due bound, siano essi interi, decimali o infiniti.

- *Infinity*: classe che rappresenta un valore infinito con il relativo segno, per essere inserito all'interno di un *bound* per generare un intervallo infinito. I metodi implementati sono i seguenti:

- `getSign()`: ritorna il segno dell'oggetto infinito;
- `setSign(char s)`: permette di settare il segno (ricevuto in input) dell'oggetto infinito;
- `isPositive()`: ritorna *true* se l'oggetto infinito ha segno positivo;
- `isNegative()`: ritorna *true* se l'oggetto infinito ha segno negativo;
- `negate()`: inverte il segno dell'oggetto infinito;
- `max(Infinity *values)`: trova il massimo in una serie di oggetti infiniti ricevuti come parametri;
- `min(Infinity *values)`: trova il minimo in una serie di oggetti infiniti ricevuti come parametri;
- override di `operator+`, `operator-`, `operator*`, `operator/`: effettuano le operazioni tra due oggetti infiniti gestendo tutti i casi particolari che si possono verificare;

- override di `operator<`, `operator<=`, `operator>`, `operator>=`, `operator==`, `operator!=`: operano i confronti tra due oggetti infiniti gestendo tutti i casi particolari che si possono verificare.
- *UndefinedOperationException*: classe che rappresenta l'eccezione per le operazioni non definite ed estende la classe *exception*. È composta solamente dall'override del metodo `what()` per indicare la stringa di errore da stampare a video.

Sono stati inoltre creati due *namespace* per creare un container astratto per individuare due gruppi logici all'interno del progetto:

- *domain*: rappresenta il gruppo logico del dominio degli intervalli contenente le relative classi implementate: *Bound*, *Infinity*, *AbstractInterval*, *IntegerAbstractInterval*, *FloatingPointAbstractInterval*;
- *exception*: rappresenta il gruppo logico delle definizioni delle eccezioni lanciate nelle classi del dominio e contiene la classe *UndefinedOperationException*.

Linguaggio *Toy*

Per provare in modo pratico la libreria del dominio astratto degli intervalli, abbiamo definito un semplice linguaggio chiamato *Toy* (simile a quello utilizzato durante le lezioni del corso), di cui abbiamo definito la seguente sintassi.

```

<program>      ::= {<statement>\n}*
<statement>    ::= <assignment> | <conditional> | <loop>
<assignment>  ::= <identifier> = <expression>
<conditional> ::= if <condition>\n {<assignment>\n}* endif
<loop>        ::= while <condition>\n {<assignment>\n}* endwhile
<expression>  ::= <value> | <value> <operator> <value>
<value>       ::= <identifier> | <number> | -<number>
<condition>   ::= <identifier> <cmp> <number> | <boolean>
<boolean>     ::= true | false
<operator>    ::= + | - | * | /
<identifier>  ::= <letter> <id>*
<cmp>         ::= <= | >= | < | >
<id>          ::= <letter> | <digit>
<number>      ::= <digit>+ | <digit>+.<digit>+
<letter>      ::= a | b | ... | z | A | B | ... | Z
<digit>       ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

```

Dalla sintassi appena descritta si possono notare le seguenti particolarità del linguaggio *Toy*:

- ogni keyword e operatore devono essere separati dagli altri attraverso uno spazio;
- ogni statement deve terminare con il carattere di terminazione della riga;
- non sono ammessi condizionali e loop in cascata;
- non sono ammessi condizionali annidati nei loop e viceversa;
- non sono ammessi confronti tra sole variabili e soli numeri;
- gli statement *if* e *while* non possono contenere più di una condizione da verificare;
- l'indentazione può essere scelta dal programmatore;
- una variabile non può essere dichiarata senza un valore di assegnamento.

Interprete per il linguaggio

Un interprete è un programma in grado di eseguire altri programmi a partire direttamente dal relativo codice sorgente.

Lo scopo è quindi eseguire un programma, scritto nel linguaggio *Toy*, in un linguaggio di alto livello (*C++*), senza effettuare la compilazione dello stesso.

La differenza tra un interprete ed un compilatore si può notare dal fatto che il primo opera traducendo una riga del programma alla volta ed eseguendola immediatamente, il secondo invece trasforma il codice sorgente in linguaggio macchina, creando un file eseguibile.

In generale, uno dei principali vantaggi è che un linguaggio interpretato è relativamente indipendente dalla macchina e dal linguaggio in cui è scritto l'interprete, per cui è sufficiente riscrivere solo l'interprete per portare un sistema software su un'altra piattaforma.

L'obiettivo ora consiste nell'implementazione di un *interprete* che utilizza la libreria sviluppata per effettuare l'analisi di un programma scritto nel linguaggio *Toy*, definito in precedenza.

Questo interprete esegue in maniera ordinata le seguenti fasi di analisi:

1. cerca il programma scritto nel file *input.txt* con la sintassi del linguaggio *Toy* (situato nella directory */build* del progetto *ais_interpreter*);
2. legge il file riga per riga, riconoscendone lo statement contenuto (assegnamento, *if* o *while*);
3. dopo aver individuato lo statement, apporta le modifiche alle variabili del programma, generando i rispettivi intervalli;
4. modifica gli intervalli attraverso le operazioni definite come metodi della libreria;
5. stampa a video una tabella indicando gli intervalli relativi a ciascuna variabile inizializzata nel programma oggetto (nome, *lower bound* e *upper bound*).

Per svolgere le cinque fasi appena definite, l'interprete fa uso dei seguenti metodi:

- **main()**: cerca il file *input.txt*, lo legge riga per riga e per ciascuna invoca le funzioni **trim()** e **split()** per identificare lo statement corrispondente. Successivamente invoca la funzione di gestione di quest'ultimo. Alla fine dell'analisi stampa a video gli intervalli risultanti, corrispondenti a ciascuna variabile del programma;
- **trim()**: prende in input le righe del file contenente il programma ed elimina gli spazi in testa (indentazione) ed in coda ad essa;
- **split()**: prende in input la riga e la divide per spazi in più *keyword*;
- **gestione_assegnamento()**: gestisce tutti i tipi di assegnamento:
 - assegnamento semplice;
 - assegnamento complesso: contiene un'operazione aritmetica nel *right-hand* gestita con la funzione **calculator()**;
- **gestione_if()**: gestisce lo statement *if* invocando **fun()** per verificarne la condizione;
- **gestione_while()**: gestisce lo statement *while* invocando la funzione **fun()** per la verifica della clausola condizionale;
- **calculator()**: gestisce le operazioni aritmetiche tra intervalli (mediante la libreria);
- **fun()**: verifica la clausola condizionale in uno statement *if* oppure *while*; se la condizione è falsa invoca la funzione **find()**, altrimenti prosegue l'esecuzione;
- **find()**: funzione invocata quando la clausola condizionale degli statement *if* oppure *while* è falsa. Permette di saltare il corpo del relativo statement, ignorandone i comandi; il controllo riprenderà dalla prima riga seguente ad esso.

Test

Dopo aver implementato il codice relativo alla libreria del dominio degli intervalli, sono stati scritti i file di test (nella directory *tests*), corrispondenti a ciascuna classe implementata, basandosi sulla libreria *Catch2* (proposta dal docente del corso di programmazione *C++*).

L'obiettivo di ogni test è quello di fornire una copertura del codice il più vicino possibile al 100%, per individuare tutti i possibili comportamenti particolari dell'intervallo.

Per ottenere ciò, ad ogni funzione di ciascuna classe sono stati forniti degli input che permettessero di verificare la validità di ogni output ottenuto dall'esecuzione delle varie tracce del programma.

Questi sono stati utili anche per verificare la corretta implementazione della libreria anche nei casi limiti delle operazioni tra intervalli.

Per compilare ed avviare i test è sufficiente eseguire i seguenti comandi da terminale (spostandosi all'interno della directory */build*):

- `make` (per compilare l'intero codice);
- `make test` (per eseguire i test implementati).

Documentazione

Per ogni classe e per ciascuna funzione in esso contenuta, è stata scritta la documentazione corrispondente attraverso lo standard tool *Doxygen*. Questa, presente nel codice sorgente, una volta compilata ed avviata, verrà generata la directory */doc* contenente la documentazione *HTML*.

Per ciascun metodo viene definito quanto segue:

- una breve descrizione (`@brief`);
- i parametri che riceve in input, se presenti (`@param`);
- il valore di ritorno, se la funzione non è *void* (`@return`).

In cima al codice di ogni file vengono riportati: il copyright (`@copyright`), la licenza (`@license`), gli autori (`@authors`), la data di produzione (`@date`), la versione del progetto (`@version`) ed una breve descrizione della classe (`@brief`).

Per generare la documentazione è sufficiente eseguire i seguenti comandi da terminale (spostandosi all'interno della directory */build*):

- `make` (per compilare l'intero codice);
- `make doc` (genera la documentazione nella directory *doc*).

Conclusioni

Per lo sviluppo di questo progetto è stato necessario dedicare inizialmente del tempo per l'apprendimento e la familiarizzazione con il nuovo linguaggio *C++*. Ciò è stato possibile attraverso il materiale didattico fornitoci durante le lezioni frontali del corso di *Linguaggio di programmazione C++* ed alcune guide online:

- guida di *TutorialsPoint*: <https://www.tutorialspoint.com/cplusplus/index.htm>;
- slide del docente: <https://github.com/FedericoUnivr/Modern-CPP-Programming>;
- guida di *Doxygen*: <http://www.stack.nl/~dimitri/doxygen/>;
- guida *Catch2*: <https://github.com/catchorg/Catch2/blob/master/docs/tutorial.md>.

Per quanto riguarda analisi statica, interpretazione astratta e dominio degli Intervalli, è stato necessario un ripasso attraverso il materiale del docente e qualche approfondimento mediante documentazioni online:

- slide del docente;
- approfondimenti: <http://www.di.ens.fr/~cousot/AI/>.

Durante l'implementazioni abbiamo incontrato piccoli ostacoli però sono stati presto superati senza eccessive perdite di tempo:

- scarsa confidenza con il nuovo linguaggio a causa delle sue differenze rispetto ad altri linguaggi a noi più noti;
- apprendimento della libreria *Catch2* per la scrittura di test;
- stesura del file *CMakeLists.txt* per consentire la compilazione ed esecuzione del codice sorgente, dei test e della documentazione in maniera automatica;
- ripasso per la gestione di casi limite durante le operazioni tra intervalli;
- ripasso riguardo la teoria per la costruzione di un interprete.

L'implementazione di questo progetto ci è stata utile per effettuare un ripasso di alcuni argomenti e per approfondire, osservandone il comportamento in ambito pratico, il funzionamento del dominio trattato durante le lezioni del corso di *Analisi dei sistemi informatici*.

Ci ha inoltre permesso di mettere in pratica le nostre conoscenze ottenute durante il percorso di studi triennale per la stesura di un interprete per un semplice linguaggio.

Abbiamo avuto modo di imparare un nuovo linguaggio di programmazione approcciandolo subito attraverso la stesura di un progetto utile ed esteso anziché mediante banali esercizi.

Ci siamo occupati di organizzare il codice in classi e documentarlo, per migliorarne la leggibilità. È stato prodotto un elevato numero di test per dare una buona copertura di tutto il codice sorgente al fine di evitare indesiderate situazioni che possano presentare errori a run-time.