Università degli Studi di Verona

DIPARTIMENTO DI INFORMATICA
Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

Analisi dei Sistemi Informatici

Dominio Astratto degli Intervalli

Autori:
Marco Colognese VR423791
Mattia Rossini VR423614

Anno Accademico 2017/2018

Indice

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*.

Background

I programmi informatici spesso contengono errori. Eseguire un programma con dei *bug* può portare a conseguenze indesiderate. Testare il codice non è sufficiente, poiché è possibile ricoprire solo una parte delle possibili tracce di esecuzione. Quando sono richieste garanzie importanti riguardo la correttezza di un programma, è necessario far uso dell'analisi statica.

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, dunque si introduce qualche grado di approssimazione. Se denotiamo con $[\![\cdot]\!]$ il comportamento concreto del programma e con $[\![\cdot]\!]^\sharp$ il comportamento approssimato, otteniamo che:

$$[\![\mathrm{P}]\!] \subseteq [\![\mathrm{P}]\!]^\sharp$$

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 un *framework* per l'analisi statica che definisce un'astrazione corretta per la semantica concreta di un programma (potenzialmente infinita).
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*: $\alpha : C \to A$;
- eventualmente una *funzione di concretizzazione*: $\gamma : A \to C$.

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* $\bot$ 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

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

Semantica astratta dei comandi

$D = (\mathbb{V}ar \to \mathbb{I}) \cup \{\bot\}$

$$[\![;]\!]^\sharp D = D$$

$$[\![NonZero(e)]\!]^\sharp D = \begin{cases} \bot & \text{se } [0,0] = [\![e]\!]^\sharp D \\ D & \text{se } [0,0] \neq [\![e]\!]^\sharp D \end{cases}$$

$$[\![Zero(e)]\!]^\sharp D = \begin{cases} D & \text{se } [0,0] \subseteq [\![e]\!]^\sharp D \\ \bot & \text{se } [0,0] \not\subseteq [\![e]\!]^\sharp D \end{cases}$$

$$[\![x \leftarrow e]\!]^\sharp D = D[x \mapsto [\![e]\!]^\sharp D]$$

$$[\![x \leftarrow M[e]]\!]^\sharp D = D[x \mapsto [-\infty, +\infty]]$$

$$[\![M[e_1] \leftarrow e_2]\!]^\sharp D = D$$

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 $[\nabla : P \times P \to P]$ 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 $x_0 \leq x_1 \leq ... \leq x_n$ la catena definita come $y_0 = x_0, ..., y_{n+1} = y_n \nabla x_{n+1}$ 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-puntofisso di F^∇, ovvero una sovra-approssimazione del punto fisso più piccolo di f.
Il widening sul dominio degli intervalli funziona come segue:

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

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 $\triangle : P \times P \to P$ tale che:

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

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

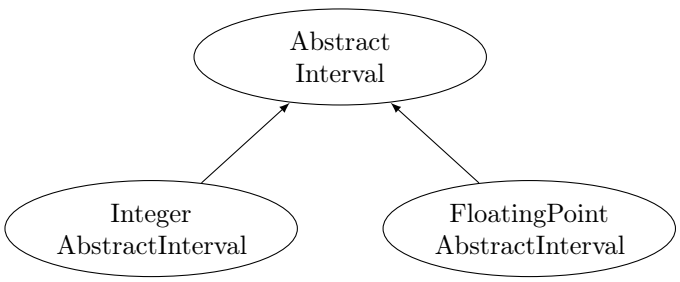
$$[a,b] \ \triangle \ [c,d] = [e,f] \qquad \text{t.c.} \qquad e = \begin{cases} c & \text{se } a = -\infty \\ a & \text{altrimenti} \end{cases} \qquad f = \begin{cases} d & \text{se } b = +\infty \\ b & \text{altrimenti} \end{cases}$$

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();`
  - `setLowerBound(Bound object);`
  - `getUpperBound();`
  - `setUpperBound(Bound object);`
  - `isFinite();`
  - `isContainedIn(AbstractInterval &i);`
  - `width();`
  - `intersects(AbstractInterval &i).`
- *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==`;
  - `lub(IntegerAbstractInterval &i, IntegerAbstractInterval &ii);`
  - `negate();`
  - `widening(IntegerAbstractInterval &i);`
  - `narrowing(IntegerAbstractInterval &i).`
- *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==`;
  - `lub(FloatingPointAbstractInterval &i, FloatingPointAbstractInterval &ii);`
  - `negate();`
  - `widening(FloatingPointAbstractInterval &i);`
  - `narrowing(FloatingPointAbstractInterval &i).`



- *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()`;
  - `getInfinityValue()`;
  - `getFloatValue()`;
  - `getValue()`;
  - `is_infinity()`;
  - `is_integer()`;
  - `is_float()`;
  - `revertSign()`;
  - override di `operator+`, `operator-`, `operator*`, `operator/`;
  - override di `operator<`, `operator<=`, `operator>`, `operator>=`, `operator==`, `operator!=`.
- *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()`;
  - `setSign(char s)`;
  - `isPositive()`;
  - `isNegative()`;
  - `negate()`;
  - `max(Infinity *values)`;
  - `min(Infinity *values)`;
  - override di `operator+`, `operator-`, `operator*`, `operator/`;
  - override di `operator<`, `operator<=`, `operator>`, `operator>=`, `operator==`, `operator!=`.
- *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.

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>     ::= <identidier> <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

L'obiettivo ora consiste nell'implementazione di un *interprete* che utlizza 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 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.