Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea in Informatica

DEBUGGING REVERSIBILE DI UN FRAMMENTO DEL LINGUAGGIO C

Relatore:
Prof.
Ivan Lanese

Presentata da:
Davide Roncuzzi

Sessione III
Anno Accademico 2019/20

Indice

Introduzione **1**

1 Background **3**

1.1 ANTLR 3

1.1.1 Grammatica e Albero 5

1.2 Debugging reversibile 8

2 Funzionalità del debugger **11**

2.1 Primo Esempio 12

2.2 Secondo Esempio 16

3 Struttura del Debugger **19**

3.1 Riconoscimento del linguaggio C 19

3.1.1 Rappresentazione del codice tramite l'albero sintattico 20

3.2 Esecuzione del codice 22

3.2.1 Variabili e Scope 23

3.2.2 Funzioni 24

4 Reversibilità **27**

4.1 History 27

4.1.1 Memoria delle istruzioni 28

4.1.2 Ripristino dello stato precedente 30

5 Conclusioni **35**

Bibliography **37**

Introduzione

La scrittura di codice software è un'attivitá non semplice, che porta spesso a fare errori durante la sua stesura, errori che, secondo studi [1], finirebbero per raddoppiare il tempo totale che verrebbe altrimenti impiegato.

Per questo motivo, vengono spesso utilizzati programmi appositi, detti debugger, per agevolare l'individuazione degli errori, detti appunto bug in inglese.

I debugger aiutano il programmatore fornendogli uno strumento tramite il quale analizzare più dettagliatamente il codice, permettendo spesso di controllare l'esecuzione del programma un passo alla volta e di vedere i valori precisamente assunti dalle variabili in ogni momento dell'esecuzione.

Tradizionalmente, la fase di debugging, avviene tramite il cosiddetto debugging ciclico. Quando ci si accorge di un comportamento non opportuno del programma, si utilizzano punti di stop, detti breakpoint, per fermare l'esecuzione alle righe di codice che si sospetta generino il malfunzionamento. Questo procedimento può essere lento, poichè è difficile posizionare i breakpoint nei punti precisi dove avvengono i bug. Di conseguenza, spesso è necessario rieseguire più volte il programma con breakpoint posizionati in punti diversi per sapere dove riparare il problema.

La computazione reversibile, che consiste nel poter rivolgere l'esecuzione del programma all'indietro, oltre che avanti, offre all'utente una padronanza maggiore e, riprendendo gli studi fatti sull'argomento[1], permette di risparmiare fino al 25% del tempo sulla fase di debugging. Tuttavia l'implementazione della reversibilità in un debugger risulta non banale, come verrà

mostrato nei capitoli successivi.

Con questo progetto descrivo un debugger testuale con reversibilità sequenziale per un frammento del linguaggio C, mostrando i metodi utilizzati per il riconoscimento e l'analisi del linguaggio, le struture dati del debugger e come viene utilizzato. In seguito, approfondisco il concetto di reversibilità e la sua implementazione nel debugger.

Capitolo 1

Background

1.1 ANTLR

La necessità del debugger di riconoscere il linguaggio C preso in input viene soddisfatta tramite il software ANTLR (Another Tool for Language Recognition) [10]. ANTLR, seguendo regole imposte da una grammatica, divide il codice in input in una serie di costrutti letterali più semplici e ne verifica la correttezza (parsing). Nel nostro caso, l'input sarà diviso nelle singole istruzioni, che verranno ramificate nei loro singoli comandi.

Questa divisione dell'input viene effettuata da un analizzatore lessicale (lexer) che viene generato da ANTLR a partire dalla grammmatica. Inoltre, lo scopo del lexer è quello di associare un token di riconoscimento ad ogni costrutto che viene individuato.

In particolare viene utilizzata una grammatica libera dal contesto (context-free), dove ogni regola sintattica è espressa sotto forma di derivazione da un simbolo non terminale posto a sinistra a uno o più simboli, terminali o non, a destra. Un simbolo non terminale è inteso come un simbolo che può essere espanso in ulteriori costrutti terminali, o non, secondo l'insieme di regole definite dalla grammatica.

Nel caso di ANTLR, queste grammatiche vengono ampliate tramite ulteriori costrutti sintattici e semantici[13], i quali hanno lo scopo di agevolare

il ruolo del parser nel riconoscimento del linguaggio. I costrutti semantici addizionali permettono al parser di risolvere alcuni tipi di ambiguità grazie alla costruzione a runtime di token diversi per costrutti simili [12].

Formalmente la grammatica con predicati viene descritta come una setupla G = (N, T, P, S, Π, M):

- N l'insieme dei nonterminali (regole)
- T l'insieme dei terminali (token)
- P l'insieme delle produzioni
- S ∈ N simbolo iniziale
- Π insieme di predicati semantici
- M insieme di azioni che possono modificare lo stato

L'analisi sulle strutture dell'input viene effettuata tramite un parser LL(*) [12]; il suo nome si spiega dal modo in cui analizza la sintassi. Infatti, esso ha un ordine di analisi da sinistra verso destra (**L**eft to right), passando alla derivazione sintattica più a sinistra dell'albero sintattico (**L**eftmost derivation).

In aggiunta, al momento di decidere la strada da percorrere, viene utilizzato un lookahead di dimensione al massimo pari all'input rimanente. Con il termine lookahead si intende il numero di token sintattici successivi che vengono guardati per scegliere la produzione corretta della grammatica. Questo comporta, quindi, una complessità del parser di $O(n^2)$ rispetto all'input. Tuttavia, nella maggior parte dei casi, viene utilizzato un lookahead di dimensione pari a uno o a due, associato a complessità lineare.

Quindi, per mantenere coerenza, la grammatica di partenza deve essere priva di ambiguità, ovvero una stringa di input non deve essere raggiungibile attraverso espansioni di regole diverse. Il percorso che viene compiuto deve quindi essere deterministico, altrimenti ANTLR prova a risolvere l'ambiguità

tramite l'ordine delle produzioni nella grammatica, che non sempre porta al risultato desiderato.

Inoltre, la grammatica deve anche essere priva di ricorsione sinistra, cioè non deve esistere un percorso a partire da un non-terminale che raggiunga una costruzione sintattica con lo stesso non-terminale posto all'estrema sinistra. Tenendo conto della tecnica di scorrimento del parser di leftmost derivation, risulta evidente come una ricorsione sinistra manderebbe lo stesso parser in un loop infinito. In ogni caso, a partire da una grammatica libera da contesto con ricorsione sinistra, è sempre possibile arrivare ad una equivalente senza.

1.1.1 Grammatica e Albero

La costruzione della grammatica [11] parte dai costrutti lessicali di base che si vogliono riconoscere. In questo caso, prendiamo come esempio i caratteri alfanumerici e numerici, che nel codice identificano variabili e valori .

```
ID  :('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')*
    ;

INT :('0' | '1'..'9' '0'..'9'*)
    ;
```

Questi vengono poi associati tra loro per regole più complesse, ad esempio la regola di assegnamento:

```
assignStat :ID EQ expr ';'
```

La regola indica come si possa riconoscere un input formato da: caratteri alfanumerici (ID), il simbolo di uguaglianza (EQ), un espressione (expr), formata da ulteriori costrutti, ed il punto e virgola. Da notare come i costrutti di base siano rappresentati con le lettere maiuscole, mentre quelli composti utilizzano lettere minuscole.

ANTLR, inoltre, tramite una serie di ulteriori regole associate alla grammatica, permette la costruzione di un albero sintattico corrispondente all'in-

put. Queste nuove regole prendono il nome di regole di riscrittura (rewrite rules) e sono nella forma:

$$\text{"rule"} \rightarrow \text{"rewrite rule"}$$

Riprendendo come esempio la regola di assegnamento sopra riportata:

```
:ID EQ expr ';' -> ^(EQ ID expr)
```

La regola di riscrittura è impostata in modo tale da rendere il primo elemento interno alle parentesi come padre degli elementi successivi. In questo caso l'albero che verrà generato avrà come radice il token EQ, mentre come figli ID ed expr.

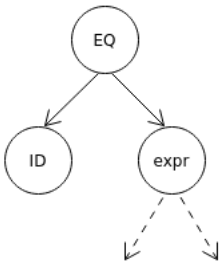

Figura 1.1: Struttura ad albero formata da ANTLR per la regola assignStat

Spesso, quando non è possibile dedurre un chiaro genitore, una regola di riscrittura deve utilizzare un token fittizio al suo posto. Con "fittizio" si intende che questo token non ha una base reale nel codice in input, viene invece formulato in base al riconoscimento della regola grammaticale e risulta necessario per identificare il ramo da cui proviene durante la scansione dell'albero sintattico.

La costruzione di questa struttura ad albero risulta essenziale per attraversare le istruzioni del linguaggio di programmazione nell'ordine corretto sia avanti che indietro.

Infatti ANTLR fornisce una struttura dati [9] per la navigazione dell'albero che permette il passaggio da un nodo all'altro tramite puntatori. Inoltre ad ogni nodo è associato un token che ne permette l'identificazione a partire dalle regole di riscrittura della grammatica.

Il token viene rappresetato da ANTLR come un ulteriore struttura, la quale, oltre a contenere l'effettivo token sintattico associato, fornisce anche i dati del relativo file di input per l'identificazione della riga da cui proviene il token, agevolando lo sviluppo delle funzionalità previste dal debugger.

1.2 Debugging reversibile

Con reversibilità, si intende la possibilità di riportare l'esecuzione del programma ad un passo precedente rispetto al punto a cui si è arrivati, di fatto annullando l'esecuzione di una serie di istruzioni. Queste potranno essere ripercorse a discrezione dell'utente per una serie di scopi, tra cui il ritrovamento di eventuali errori presenti nel codice.

Questo processo prende il nome di backtracking e viene convalidato a livello teorico dal "loop lemma", il quale indica che eseguendo un'azione per poi annullarla si ritorna sempre al punto di partenza [3] [6].

Questo concetto di reversibilità è stato velocemento associato al debugging, in contrapposizione con il tradizionale debugging ciclico che comporta numerose ripetizioni dell'esecuzione del programma per identificarne i malfunzionamenti.

Inizialmente, era nata una via intermedia tra i due tipi di debugging, chiamata record-replay, rintracciabile sin dagli anni '80 [7]. In questa modalità, viene salvata l'esecuzione del programma a partire da certi stati (record), per poi essere ripetuta, a partire da essi, a discrezione dell'utente (replay).

Sostanzialmente, in questo caso, l'esecuzione del programma viene sempre svolta in avanti, partendo da determinati checkpoint, senza tornare indietro. Quindi non può essere intesa come una tecnica propriamente reversibile, anche se vicina ad essa per concetto. Invece, il primo debugger reversibile che permette effettivamente di andare avanti ed indietro nell'ordine di esecuzione del programma è stato sviluppato durante i primi anni 2000[4] [16].

Tra i debugger moderni che sfruttano la reversibilità si evidenziano WinDgb[2] e il software open source GDB[5]. Quest'ultimo, in particolare, è stato utilizzato spesso per lo sviluppo di questo progetto e alcune funzionalità sono state sviluppate ispirandosi a quelle di questo debugger. Il debugger mostrato in questa tesi, in aggiunta, mette le basi per un futuro supporto della programmazione concorrente.

I principali metodi per l'implementazione della reversibilità sono associati al modo in cui si mantengono le informazioni aggiuntive necessarie per

ripristinare gli stati precedenti.

La tecnica trace-based consiste nell'utilizzare un registro (log) per memorizzare completamente l'esecuzione di ogni istruzione del programma a partire dal linguaggio macchina [4]. Questo metodo comporterebbe una certa laboriosità a livello software, per questo motivo le sue prime proposte implementative utilizzavano un hardware apposito per la memorizzazione degli eventi dell'esecuzione, le cui informazioni venivano in seguito interpretate dal software del debugger. Di questo tipo erano i primi debugger reversibili entrati in commercio nella metà degli anni duemila[8].

Un secondo metodo per implementare la reversibilità, consiste nella ricostruzione dello stato a partire da stati precedenti da cui si può raggiungere il punto voluto, salvando solamente le informazioni strettamente necessarie.

Infine, l'approccio che viene utilizzato per il debugger descritto in questo elaborato, consiste nel raggiungere l'istruzione desiderata tramite l'annullamento delle istruzioni successive ad essa, mantenendo a run-time i dati necessari per la ricostituzione degli stati.

Capitolo 2

Funzionalità del debugger

Descrivo ora le caratteristiche del debugger sviluppato, le singole funzionalità ed in seguito degli esempi del suo funzionamento. Esso utilizza la linea di comando in ambiente Linux. Il programma su cui agirà il debugger deve essere scritto in un file dal nome "input.c" posto all'interno della stessa cartella dell'eseguibile[15]. Altrimenti, un file specifico può essere passato come parametro dell'eseguibile.

Per utilizzare il debugger, bisogna semplicemente lanciare l'eseguibile digitando il comando ./m da terminale dalla cartella in cui si trova. Immediatamente, il debugger comincerà ad eseguire il programma in input, eseguendo la prima istruzione per poi fermarsi richiedendo l'input dell'utente.

Il debugger fornisce la possibilità di spostarsi in avanti all'interno del programma per una o più istruzioni alla volta, oppure eseguendo il programma normalmente. In quest'ultimo caso, si possono utilizzare breakpoint per fermarsi a predefinite righe del codice in input.

Quando l'esecuzione del programma viene interrotta dal debugger, viene mostrata all'utente l'ultima istruzione eseguita ed un input testuale può essere utilizzato per scegliere come proseguire.

A questo punto viene data la possibilità all'utente di vedere i valori corrispondenti alle variabili in questo punto del programma. In particolare, l'utente può anche invertire il senso di esecuzione del programma, ritornando

all'istruzione precedente rispetto al punto in cui si trova.

In seguito sono descritti i singoli comandi in una lista:

- "h" lista dei comandi
- "n" avanti di una singola istruzione
- "n" + *int* andare avanti di *int* istruzioni alla volta
- "p" tornare indietro di un'istruzione
- "print" + *var* per stampare il tipo e valore della variabile *var*
- "break" + *line* impostare un breakpoint alla linea *line*
- "remove"+ *line* togliere un breakpoint dalla linea *line*
- "removeall" togliere tutti i breakpoint
- "run" per lanciare il programma normalmente, interrompendosi solo per gli eventuali breakpoint

2.1 Primo Esempio

Illustriamo ora le funzionalità con un esempio; il seguente programma dovrebbe calcolare e stampare i fattori primi di un numero. Con il primo ciclo vengono controllati i divisori del numero scelto, mentre con il secondo ciclo annidato si controlla se si tratta di numeri primi, contando quanti divisori hanno.

Esempio di codice corretto

```
1    int Number = 10;
2    int i = 1;
3    int j;
4    int Count;
5
```

```
6     while (i <= Number){
7         Count = 0;
8         if(Number % i == 0){                    //factor-check
9             j = 1;
10            while(j <= i){                        //prime-check
11                if(i % j == 0){
12                    Count = Count + 1;
13                }
14            j = j + 1;
15            }
16            if(Count == 2){
17                printf("%d", i);
18            }
19        }
20        i = i + 1;
21        }
```

Consideriamo ora un errore comune nella programmazione, ovvero quello di sbagliarsi mentalmente sui casi limite, al posto del minore o uguale come condizione di controllo all'interno del ciclo while a riga 10, mettiamo come condizione semplicemente il minore, di fatto facendo sfuggire un singolo caso al controllo.

	corretto		*errato*
10	`while(j <= i){`	–>	`while(j < i){`

Per questo esempio utilizziamo come valore di input 10. Il programma dovrebbe stampare quindi 2 e 5, gli unici divisori primi di 10.

Tuttavia, eseguendo il programma, il programmatore si accorge che esso termina senza stampare nulla. Il conseguente ragionamento logico è quello

di controllare la riga che gestisce le stampe, ovvero la riga 17 e la condizione che viene controllata per arrivare a riga 16, ipotizzando un possibile valore errato della variabile Count.

```
line 1:     int Number = 10

enter a command (h for commands)
break 16
breakpoint set to line 16
enter a command (h for commands)
run
```

Figura 2.1: debugger con il comando break per inserire un breakpoint

Questa volta viene eseguito il programma tramite il debugger e viene messo un breakpoint alla riga 16. In seguito all'avvio del programma, si nota che il breakpoint viene raggiunto e si può controllare se la condizione deve essere soddisfatta in questo primo ciclo del while. Il programmatore, utilizzando il comando print, stampa il valore della variabile i, che indica i divisori 10. In questo caso, i ha valore 1, che essendo un divisore primo non significativo non deve essere stampato. Il valore di Count, che risulta essere 0, riflette correttamente questo aspetto.

Continuando ad usare il comando run, si può passare al momento in cui il ciclo ripercorre la linea 16 per la seconda volta. In questo caso, il valore di Count risulta essere 1, ed il valore del divisore i invece è 2. Il tester riconosce che il valore 2 è un divisore primo significativo di dieci, che di conseguenza dovrebbe essere stampato.

Ora, appurato che il valore di Count è errato, come step successivo il programmatore vuole controllare le righe che vanno a modificare il suo valore, cioè riga 12 ed in particolare la condizione a riga 11.

Sfruttando la proprietà di reversibilità del debugger si può facilmente tornare ai passaggi precedenti, invece che dover far ripartire il programma con un nuovo breakpoint. Utilizzando il comando "p" con in mente l'idea di raggiungere l'istruzione a riga 11, il debugger mostra, come istruzione

```
line 16: if(Count == 2)

enter a command (h for commands)
print i

variable i:
 type: int
 value 2

enter a command (h for commands)
print Count

variable Count:
 type: int
 value 1
```

Figura 2.2: debugger con il comando di stampa "print"

precedente, l'ultima condizione che ha controllato il while prima di uscire dal suo ciclo.

A questo punto, sempre utilizzando il comando "print", è facile vedere come nella condizione di controllo del while sia j che i abbiano come valore 2. Con questi valori l'istruzione while dovrebbe, invece, continuare il suo ciclo. Questo permette al programmatore di capire che questo caso era sfuggito alla condizione di guardia del ciclo while (riga 10) che non teneva conto del caso appena citato.

```
enter a command (h for commands)
p
line 10:        while(j < i)
```

Figura 2.3: debugger con il comando "p" per tornare all'istruzione precedente

Dopo aver modificato il minore a riga 10 con il minore o uguale e facendo ripartire il programma, le stampe dei valori primi risultano in maniera corretta. Si noti come, con questo procedimento, il programmatore non ha avuto bisogno di eseguire più di una volta il programma prima di modificare il codice e trovare l'errore.

2.2 Secondo Esempio

Dopo aver illustrato il processo per trovare errori nel codice con l'esempio precedente, utilizzo un esempio tecnico per mostrare le caratteristiche dei puntatori che vengono supportate.

Consideriamo il seguente programma che scambia i valori di due variabili tramite puntatori:

```
1        int* a;
2        int* b;
3        int x = 5;
4        int y = 6;
5        int temp;
6
7        a = &x;
8        b = &y;
9
10       temp = *b;
11       *b = *a;
12       *a = temp;
13
14       printf("%d", x);
15       printf("%d", y);
```

Nell'esempio vengono mostrate esaustivamente le operazioni possibili su puntatori. In particolare come vengono utilizzati gli operatori di referenziazione (&) e di dereferenziazione (*).

```
line 7: a = &x

line 8: b = &y

line 10: temp = *b

line 11: *b = *a

line 12: *a = temp

line 14: printf("%d", x)

6
line 15: printf("%d", y)

5
```

Figura 2.4: Debugger con input il secondo esempio

In particolare, se viene richiesto di stampare il valore di un puntatore, il debugger stampa il tipo, il nome della variabile a cui punta e il suo valore (fig. 2.5).

```
next line:
line 8: b = &y

enter a command (h for commands)
print b
 int pointer pointing to variable: y
  y = 6
enter a command (h for commands)
```

Figura 2.5: debugger in seguito al comando print di un puntatore

Capitolo 3

Struttura del Debugger

In questo capitolo descrivo come il debugger utilizza il codice in input, di come ne viene simulata l'esecuzione trattando le strutture in maniera generale per poi essere riprese nel dettaglio nel capitolo successivo.

Il debugger, situato in una repository github[14] è scritto anch'esso nel linguaggio C. Questo permette una certa corrispondenza tra il codice dato in input e le strutture utilizzate dal debugger a runtime. Sono state scritte autonomamente circa 1500 righe di codice, a cui sono stati associati i file generati da ANTLR a partire dalla grammatica. Quest'ultima è partita dal riconoscimento di variabili e costrutti alfanumerici, per poi essere ampliata con i costrutti descritti in seguito.

Al progetto è stata associata la licenza MIT per il software libero.

3.1 Riconoscimento del linguaggio C

Il linguaggio che riconosce il debugger è un frammento del linguaggio C che comprende le seguenti funzionalità

- Tipi int char e void
- assegnamento
- espressioni aritmetiche

- istruzioni condizionali if-else
- espressioni relazionali
- istruzioni cicliche while
- puntatori a tipi int e char
- definizioni e chiamate di funzioni

Da notare come l'utilizzo delle funzioni avviene come nel linguaggio C, ovvero, per potersi avvalere di una funzione, la sua definizione deve essere posta prima del suo effettivo utilizzo. Una versione della funzione printf è implementata all'interno del debugger di default. Essa può essere utilizzata per stampare una singola variabile, come nell'esempio successivo, o una stringa.

Inoltre, nonostante non sia supportata allocazione dinamica della memoria, ad esempio tramite la funzione malloc(), sono supportati puntatori verso int e char e le loro operazioni unarie (*, &).

Questo frammento del linguaggio C rimane comunque significativo per lo studio in quanto permette di scrivere programmi non banali come mostrato negli esempi.

Siccome il debugger utilizza il linguaggio C per simulare l'esecuzione del programma, offre anche un type-checking dinamico, in quanto le strutture tipizzate vengono istanziate con lo stesso tipo dal debugger. Se, a run time, il tipo di una variabile non corrisponde con il suo utilizzo, il programma lancerà un errore.

3.1.1 Rappresentazione del codice tramite l'albero sintattico

L'esecuzione del programma in input viene simulata dal debugger percorrendo l'albero sintattico costruito tramite ANTLR, a partire dalla grammatica ampliata dalle regole di riscrittura.

La radice assoluta dell'albero è formata un genitore fittizio e tutti i suoi figli rappresentano istruzioni diverse del programma, queste sono riconoscibili grazie ai token associati ad ogni costrutto sintattico dalle rewrite rules.

Per visualizzare la costruzione dell'albero, mostro come esempio il seguente frammento di codice, riguardante il costrutto condizionale if-else:

```
if(a == b){
        a = a / 2;
}
else {
        b = b / 3;
}
```

Queste righe di codice sono riconosciute dalle seguenti regole della grammatica:

```
ifStatement
  :  ifStat elseStat? -> ^(IF_STAT ifStat elseStat?)
  ;
ifStat
  :  'if' '('condExpr ')' block -> ^(COND condExpr block)
  ;
elseStat
  :  'else' block -> ^(ELSE_STAT block)
  ;
```

Il sottoalbero generato è mostrato in figura 3.1 e la sua radice è formata dal token fittizio IF_STAT, il quale viene utilizzato per identificare il costrutto if-else quando viene raggiunto dall'analizzatore lessicale, ed in seguito dal debugger.

I suoi due figli sono anch'essi token fittizi, i quali identificano altri due costrutti non banali. Il primo, COND, è genitore a sua volta della condizione di guardia dell'if e del blocco di codice (BLOCK) di quest'ultimo. Il secondo token figlio, ELSE_STAT, è invece genitore del corpo del ramo else.

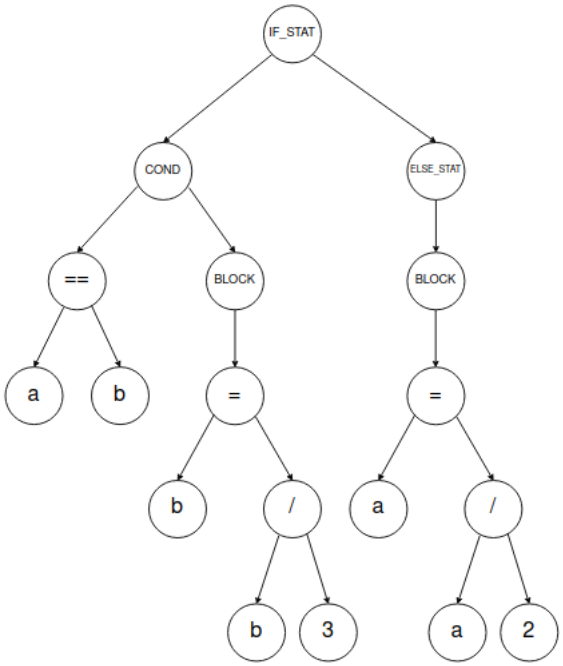

Figura 3.1: Albero generato dal costrutto if-else

Le regole delle singole istruzioni sono, invece, costruite per avere l'operazione significativa dell'istruzione come genitore e i valori associati alle variabili come figli.

3.2 Esecuzione del codice

L'esecuzione del codice C dato in input avviene scorrendo l'albero sintattico generato da ANTLR visto nel capitolo precedente.

La visita dell'albero avviene in ordine anticipato (o preordine), e la direzione di visita può essere modificata in base a quali istruzioni vengono trovate. Nel caso della figura (3.1), quando viene raggiunto il ramo che identifica l'if (COND), viene prima esaminata la condizione per poi decidere se proseguire con il ramo if oppure con il ramo else.

3.2.1 Variabili e Scope

Quando viene istanziata una variabile nel codice in input, la stessa viene istanziata a runtime dal debugger. Per tenere conto di tutti i valori salvati, viene utilizzata una hashtable con come chiave il nome e, come campo associato ad essa, la struttura che identifica la variabile con campi associati al suo tipo e al valore (fig. 4.1); questa sarà descritta nel dettaglio nel capitolo successivo.

Siccome le variabili all'interno di un programma hanno un campo di azione differente in base alla porzione di codice in cui si trovano (scope), vengono utilizzate diverse hashtable, ognuna associata ai diversi blocchi di codice presenti nel programma.

Questi blocchi hanno una relazione di parentela fra loro; infatti i blocchi annidati fanno parte del campo di azione dei loro genitori. Allo stesso modo, le variabili definite nei genitori sono visibili ai figli e non ai blocchi esterni, mentre le variabili definite nei figli non possono essere accedute dai discendenti.

Per questo motivo, le hashtable sono collegate tra loro da una struttura ad albero che riprende la struttura dei blocchi di codice. Quindi, ogni volta che viene creato un nuovo scope, ad esempio da parentesi graffe, viene creata una nuova hashtable che viene impostata come figlia dello scope precedente. Le nuove variabili definite in questo nuovo blocco di codice vengono salvate solamente all'interno dell'hashtable figlia.

In tal modo, all'uscita dal blocco formato dalle parentesi, si ritorna, anche nel debugger, allo scope padre, dal quale non si può accedere alle variabili dichiarate nel figlio, come accade nel linguaggio C.

Per quanto riguarda l'invocazione di una variabile da uno scope figlio presente in uno scope antenato, la sua ricerca si propaga a cascata, controllando prima nello scope corrente per poi passare al genitore, fino a quando si trova la variabile cercata o si raggiunge la radice dell'albero di parentela. In quest'ultimo caso, si viene allertati dal debugger che la variabile non risulta esistere in quel campo di azione.

Come visto in precedenza per il costrutto if-else, nella gestione di istruzioni che comprendono una condizione che determina l'istruzione successiva, viene prima calcolato il risultato dell'espressione condizionale di guardia a runtime per poi intraprendere la strada corretta nell'albero sintattico.

In particolare nel caso di istruzioni cicliche tramite while, la condizione viene ricalcolata con i valori aggiornati prima di rieseguire, eventualmente, le istruzioni del suo blocco di codice.

3.2.2 Funzioni

Particolare è il caso riguardante il riconoscimento e la gestione delle funzioni. Queste, infatti, vengono definite e poi utilizzate in momenti diversi del codice, necessitando comunque del loro scope. Per raggruppare i dati necessari all'esecuzione della funzione, viene usata una struttura dati che la rappresenta (fig. 3.2). Essa viene identificata dal nome della funzione e contiene il tipo di ritorno che viene utilizzato per il controllo di tipo (type checking).

Per fare in modo che il corpo della funzione sia correttamente eseguito, viene salvato in essa anche la frazione di albero sintattico che rappresenta il codice del corpo della funzione, il quale viene identificato nel momento di definizione di quest'ultima. Inoltre, anche i parametri, assieme al loro tipo, vengono salvati in una lista sempre all'interno della struttura anche se nel momento della definizione essi non hanno un valore associato. Quest'ultimo sarà aggiunto a runtime al momento della chiamata della funzione.

Infine, la struttura dati associata alla funzione contiene una struttura scope da inizializzare, la quale al momento dell'invocazione verrà popolata dai parametri con i loro relativi valori ricevuti come input.

Quindi, il debugger risponde alla chiamata della funzione creando una nuova struttura funzione, a partire da quella creata quando è stata definita la funzione. Questa nuova struttura è relativa esclusivamente a una singola invocazione della funzione.

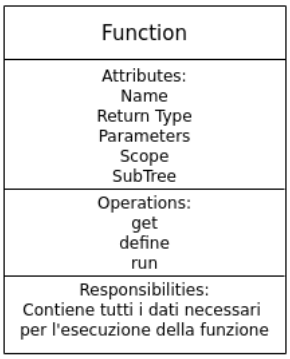

Figura 3.2: Struttura dati per l'esecuzione di una funzione

In seguito, viene eseguito il sottoalbero sintattico corrispondente, utilizzando una nuova hashtable creata a runtime come scope associato alla funzione, contenente unicamente i valori passati ai parametri nella chiamata della funzione.

Capitolo 4

Reversibilità

Vediamo ora come viene applicato nel dettaglio il concetto di reversibilità nel debugger, descrivendo in particolare la struttura di History ed in seguito un esempio.

4.1 History

Per poter effettuare l'inversione di direzione nell'esecuzione del programma, è necessario tenere traccia di una serie di elementi che sarebbero altrimenti persi dalla normale esecuzione di un programma.

Infatti, quando ad esempio una variabile viene modificata, il suo valore precedente verrebbe sovrascritto da quello nuovo, senza lasciarne traccia. Un altro esempio, riguarda il caso di costrutti condizionali, durante i quali è necessario ricordarsi quale strada è stata intrapresa.

Quindi, è opportuno l'utilizzo di una struttura aggiuntiva che tenga conto delle informazioni che verrebbero altrimenti perse, questa struttura nel campo della reversibilità prende il nome di "History" proprio per questo suo ruolo.

Di conseguenza, ad ogni passo del programma che necessita di informazioni aggiuntive per essere eseguito in senso opposto, deve essere associata una struttura che tenga in memoria queste informazioni. Queste singole strutture

che si occupano dei singoli passi del programma vengono chiamate History item e sono memorizzate con una lista bidirezionale, questa struttura ricopre il ruolo di History, appena descritto, all'interno del debugger.

4.1.1 Memoria delle istruzioni

Come visto precedentemente, per poter ripristinare istruzioni che sovrascrivono variabili, bisogna tener conto dei valori mantenuti precedentemente e di quando sono stati modificati. Per questo motivo una variabile viene rappresentata dal debugger tramite una pila (fig. 4.1), nella quale vengono salvati i valori assunti durante l'esecuzione. La modalità last in first out permette di accedere velocemente agli ultimi valori assunti dalla variabile, che sono effettivamente quelli da dover modificare in caso di ripristino di uno stato precedente.

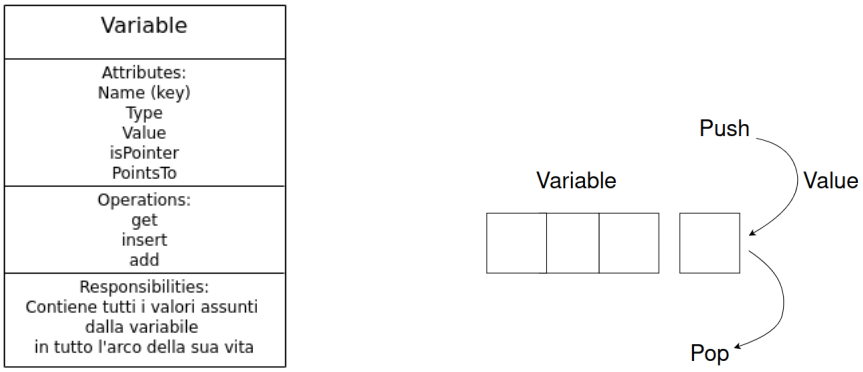

Figura 4.1: Struttura dati di una variabile

Inoltre, dentro agli History item, sempre per potersi ricondurre allo stato precedente, vengono salvate in una lista quelle variabili che sono state accedute in scrittura dal programma nel relativo passo, in modo da sapere quali variabili sono effettivamente da ripristinare ai loro valori precedenti.

Per identificare correttamente queste variabili da modificare, viene salvato all'interno di History anche lo scope relativo al passo corrente, ovvero l'hashtable che contiene tutte le variabili ed infine la sua struttura ad albero.

I puntatori vengono gestiti in maniera simile alle variabili utilizzando la stessa struttura dati, tuttavia a differenza di esse, utilizzano ulteriori campi nella struttura in maniera significativa: per indicare se l'oggetto indicato è effettivamente un puntatore (isPointer) e della lista di parametri a cui ha puntato durante l'esecuzione del programmma (fig. 4.2).

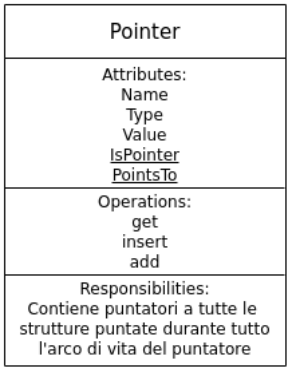

Figura 4.2: Struttura dati di un puntatore

Infine, per avere un riferimento per poter ripercorrere sequenzialmente il programma, viene salvato anche il frammento di albero sintattico associato al passo corrente, il quale viene anche utilizzato per mostrare all'utente i valori associati al nuovo stato attuale del programma come ad esempio: linea in cui si trova, istruzione ecc.

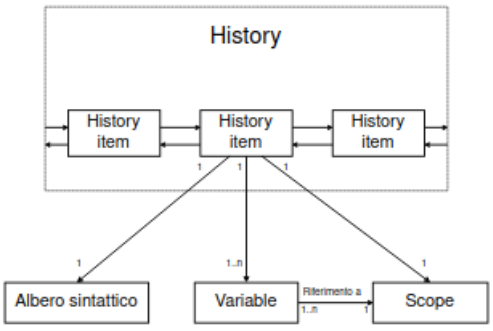

Figura 4.3: Rapporti tra History e altre strutture di dati

La figura 4.3 mostra in che modo la struttura History si rapporta con le altre strutture dati, che rapporto queste ultime hanno tra di loro. Le freccie, nell'immagine, hanno funzione di associazione e mostrano come ogni History item sia collegato al suo successore e al suo predecessore, permettendo l'orientamento all'interno dell'esecuzione del programma. In particolare, viene mostrato come ogni History item tenga conto delle informazioni sullo stato corrente tramite l'albero sintattico associato all'istruzione, le variabili che sono state modificate e lo scope ad esse associato.

4.1.2 Ripristino dello stato precedente

Vediamo ora il ruolo che svolge History nella reversibilità riprendendo l'esempio if-else del capitolo precedente, il cui albero sintattico generato è illustrato in figura 3.1.

Il debugger, scorrendo l'albero per trovare l'istruzione successiva, raggiunge il token che identifica la struttura if-else (IF_STAT). A questo punto, non c'è bisogno di registrare nessuna informazione aggiuntiva, in quanto il nodo radice è un nodo fittizio. Continuando la sua esecuzione, discende verso il primo figlio, COND, anch'esso un nodo fittizio, come visto in precedenza, che instrada verso la condizione dell'if, ovvero a == b.

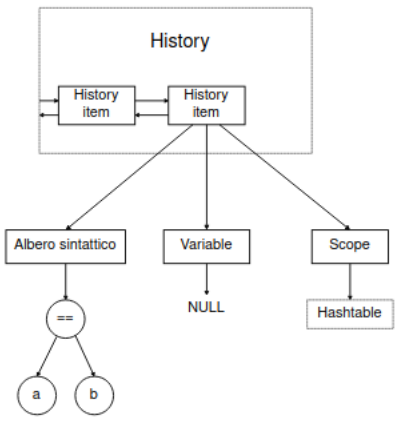

Figura 4.4: History in seguito all'istruzione a == b

Il debugger riconosce l'istruzione del programma e salva, nell'History item, i puntatori corrispondenti al sottoalbero sintattico e allo scope per identificare le variabili. Non essendoci stata nessuna modifica in memoria, non viene salvato altro. La situazione corrispondente è mostrata in figura 4.4.

Supponiamo, ora, che la condizione sia risultata falsa e che il debugger sia indirizzato sul ramo else (ELSE_STAT). Dopo aver superato anche questa volta i nodi fittizi, il debugger incontra l'istruzione a = a / 2. Dopo aver creato un nuovo History item, avviene, come in precedenza, il salvataggio del sottoalbero associato all'istruzione e il suo scope tramite puntatori. Inoltre, in questo caso, avviene un assegnamento e quindi la variabile "a" viene modificata nella memoria del programma.

Quindi è necessario avere un riferimento ai valori precedenti in caso di esecuzione in senso opposto. Il nome della variabile viene salvato nella lista apposita situata nell'History item. Ricordo che il nome rappresenta la chiave per ottenere il valore della variabile dalla hashtable che ha ruolo di scope. La struttura History si trova ora nella situazione dell'immagine 4.5

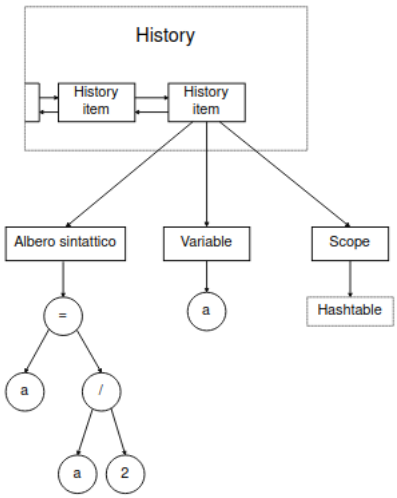

Figura 4.5: History in seguito all'istruzione a = a / 2

Simuliamo ora che l'utente voglia tornare indietro per controllare che valo-

re avesse la variabile "a" nella condizione dell'if. Dopo che il programmatore ha utilizzato il comando "p" per richiedere di invertire il senso di esecuzione, il debugger controlla l'History item, il cui passo corrente identifica appunto l'istruzione a = a / 2.

Vengono prese le variabili che sono state modificate in questo passo e viene presa la struttura dati (fig. 4.1) a loro associata dallo scope. Da questa struttura viene eliminato l'ultimo valore assegnato alla variabile tramite un'operazione pop.

A questo punto il debugger modifica il riferimento all'History item corrente tornando a quello dell'istruzione precedente.

Da questo History item, adesso diventato quello attuale, viene preso il riferimento al sottoalbero dell'istruzione a == b e, tramite il token associato ad esso, viene mostrato all'utente il punto preciso del programma cui si è arrivati tramite la stampa dei caratteri della linea e il suo numero.

Ora l'utente è effettivamente tornato al punto dell'esecuzione appena successivo all'istruzione a == b e può utilizzare i comandi di stampa per verificarne i valori.

Per poter conciliare la visita dell'albero sintattico con la possibilità di poter percorrere il programma avanti e indietro, quando l'utente richiede di ritornare ad un istruzione precedente, la struttura dell'History item corrente non viene cancellata, rimane, invece, nella lista in posizione successiva rispetto al puntatore che indica l'History item corrente.

Quando viene richiesto di tornare avanti con l'esecuzione del programma, il debugger, invece di proseguire con l'esecuzione dell'albero sintattico di partenza, controlla se sono già presenti History delle istruzioni successive, ed esegue il codice a partire dai frammenti degli alberi sintattici presenti in quest'ultime dato che in esse sono già presenti i sottoalberi corretti.

Questo risulta necessario poichè non è sempre possibile trovare l'istruzione sequenzialmente successiva partendo dal sottoalbero dell' istruzione precedente, a causa delle ramificazioni in cui il sottoalbero di partenza si

può trovare. In tal modo, viene fornita all'utente la possibilità di eseguire un numero indeterminato di avanzamenti o indietreggiamenti all'interno del codice.

Nel caso del nostro esempio, se l'utente volesse andare avanti con l'esecuzione, al posto di ricalcolare tutta la struttura IF_STAT per scegliere il ramo else, il debugger esegue l'istruzione contenuta nell'History item successivo in quanto si sa già che l'istruzione successiva sarà a = a / 2.

Capitolo 5

Conclusioni

Con questo elaborato ho descritto il processo per l'implementazione di un debugger testuale reversibile per il linguaggio C, mostrando in dettaglio le strutture dati utilizzate. Sono stati approfonditi i metodi utilizzati per l'analisi sintattica del linguaggio a partire dalla sua grammatica.

Una particolare enfasi è stata data al concetto di reversibilità ed è stato mostrato come il suo utilizzo nella fase di debugging comporti una sua accelerazione.

Questo lavoro lascia aperte numerose possibilità di ampliamenti futuri, ad esempio estendendo il linguaggio riconosciuto dal debugger con strutture (struct), oppure l'aggiunta di allocazione dinamica della memoria.

In particolare, questo lavoro, è cominciato con l'idea di aggiungere, in futuro, il supporto della programmazione concorrente, associandola alla reversibilità del debugger.

Infatti, il debugger può tenere traccia anche di quelle variabili che sono state solamente accedute in lettura, una funzionalità necessaria per la concorrenza, che invece non viene utilizzata per il solo uso sequenziale del debugger attuale. Questa funzionalità aggiuntiva serve poichè i programmi concorrenti possono accedere e modificare dati in memoria in un ordine non predefinito. Di conseguenza, per ripristinare uno stato, bisogna anche sapere che valore aveva una variabile nel momento della sua lettura.

Bibliography

[1] Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak e Tomer Katzenellenbogen. *Reversible Debugging Software: Quantify the time and cost saved using reversible debuggers*. 2013.

[2] *CppCon2017*. URL: https://github.com/CppCon/CppCon2017.

[3] Vincent Danos e Jean Krivine. "Reversible Communicating Systems". In: *CONCUR 2004 - Concurrency Theory*. A cura di Philippa Gardner e Nobuko Yoshida. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 292–307.

[4] J. Engblom. "A review of reverse debugging". In: *Proceedings of the 2012 System, Software, SoC and Silicon Debug Conference*. 2012, pp. 1–6.

[5] *GDB*. URL: https://www.gnu.org/software/gdb/documentation/.

[6] Ivan Lanese. "From Reversible Semantics to Reversible Debugging: 10th International Conference, RC 2018, Leicester, UK, September 12-14, 2018, Proceedings". In: gen. 2018, pp. 34–46. ISBN: 978-3-319-99497-0. DOI: 10.1007/978-3-319-99498-7_2.

[7] T.J. Leblanc e John Mellor-Crummey. "Debugging Parallel Programs with Instant Replay". In: *Computers, IEEE Transactions on* C-36 (mag. 1987), pp. 471–482. DOI: 10.1109/TC.1987.1676929.

[8] M. Lindahl. "The device software engineer's best friend". In: *Computer* 39.5 (2006), pp. 95–97. DOI: 10.1109/MC.2006.179.

[9] Terence Parr. *Antlr data structures for the C api*. URL: https://www.antlr3.org/api/C/index.html.

[10] Terence Parr. *Antlrv3*. URL: https://www.antlr3.org/. (accessed: 15.11.2020).

[11] Terence Parr. *The Definitive ANTLR Reference: Building Domain Specific Languages*. 2007. ISBN: 9780978739256.

[12] Terence Parr e Kathleen Fisher. "LL(*): the foundation of the ANTLR parser generator". In: vol. 46. Giu. 2011, p. 425. DOI: 10.1145/1993316.1993548.

[13] Terence Parr e Russell Quong. "Adding Semantic and Syntactic Predicates To LL(k): pred-LL(k)". In: vol. 786. Mag. 1994, pp. 263–277. ISBN: 978-3-540-57877-2. DOI: 10.1007/3-540-57877-3_18.

[14] Davide Roncuzzi. *Debugger reversibile C*. URL: https://github.com/drRiss/Debugger-reversibile-C.

[15] Davide Roncuzzi. *Main project*. URL: https://github.com/drRiss/Debugger-reversibile-C/tree/main/Antlr-C-.

[16] Shyh-Kwei Chen, W. K. Fuchs e Jen-Yao Chung. "Reversible debugging using program instrumentation". In: *IEEE Transactions on Software Engineering* 27.8 (2001), pp. 715–727. DOI: 10.1109/32.940726.