Capitolo 1

Introduzione

1.1 Analisi statica

Analisi statica (alla SWE): analisi di un programma senza l'esecuzione a *runtime*. Permette di ottenere informazioni sul programma stesso.

L'obiettivo del corso di Verifica del *software* comprende l'utilizzo della correttezza per comprendere l'esecuzione di un programma a *runtime*. Permette di ottenere la piena garanzia di correttezza del programma.
La verifica può venire fatta per qualunque versione del codice sorgente (*byte-code* o altro). Viene svolta con analizzatori di codice o moduli. Quest'ultimi effettuano una verifica progressiva, parallela alla scrittura.
Polyspace: strumento di analisi di codice statico, dimostra l'esistenza di errori a *runtime* critici. Verifica codice C, C++ o Ada.
Un problema esistente è la mancata scalabilità della verifica.
Quando si scrive un analizzatore è bene utilizzare un linguaggio di programmazione robusto (come Ada). Tuttavia la quasi assenza di persone competenti in tale ambito incide sulla difficoltà di risolvere i problemi legati agli analizzatori.

Alcuni esempi di analizzatori:

- Interproc è un analizzatore accademico che inferisce invarianti;
- Jandom è un analizzatore Java, scritto in Scala (ovvero Java funzionale avanzato).

Di recente è l'impiego degli analizzatori per gli algoritmi di *Machine Learning*.

1.2 Motivazioni

Le motivazioni che portano allo studio dell'analisi statica:

1. Fallimenti *software*: caso di Ariane 5.
   Il razzo una volta lanciato in aria si autodistrugge.
   Si era verificato un *software error*, conversione tra virgola mobile a intero, con perdita d'informazione, che ha lanciato un'eccezione non catturata

causando la chiusura di tutti i programmi del razzo. L'ultimo programma eseguito è stato l'autodistruzione.

2. *Meltdown* e *Spectre*: i processori moderni usano l'esecuzione speculativa (tecnica di otimizzazione. L'eleaboratore esegue operazioni necessarie forse solo in un secondo tempo. `https://it.wikipedia.org/wiki/Esecuzione_speculativa`.) per velocizzare il lavoro. Un *team* di ricercatori ha su tale tecnica individuato un *bug* di sicurezza, causato delle *miss prediction* che lasciavano dati sensibili all'inteno delle *cache*, innescando eventuali attacchi malevoli. L'Intel, precedentemente la pubblicazione della ricerca, avvisata dal *team*, ha mitigato il problema in modo *hardware*. Un modo per impedire la nascita di questi *bug* è svolgere già durante lo sviluppo del codice l'analisi statica.

Una buona tecnica, per punti, che permette di prevenire fallimenti del codice è la seguente:

- Scegliere un buon linguaggio;
- Svolgere progettazione;
- Effettuare *code testing*;
- Utilizzare un metodo formale di analisi statica. Questo permette la totale garanzia della correttezza del codice prodotto.

Capitolo 2

Programmazione Semantica

La semantica è un asse fondamentale della *Programming Language Theory* e si occupa di valutare una sequenza di simboli con regole (sintassi) e i meccanismi di grammatica in modo da mostrare il significato del calcolo. In sostanza permette di realizzare un modello.
Il caso di *parser* che rigetta il programma è un problema di sintassi, non di nostra competenza.

```
if 1=1 then S1 else S2
```
semanticamente equivale a scrivere $S1$.

Perchè fare semantica?

- Per dare una specifica su un linguaggio;
- Come supporto del *testing*;
- Permette di avere le basi per un prototipo (per interpreti e compilatori).

CompCert è un compilatore per C, con impatto pratico e industriale certificato. Il codice che produce (oggetto) è equivalente al codice sorgente. Per questo c'è un modello sia per il codice oggetto che per il codice sorgente. CompCert usa il dimostratore Coq, che permette di verificare teoremi.
Il codice generato da CompCert è vicino al GCC, ritenuto il più efficiente ed è a licenza per usi commerciali e *open-source* se per usi personali.

Il linguaggio che useremo un questo corso è *Turing* completo e non impiegeremo il tipo puntatore in modo da non fare analisi statica all'interno della memoria.

Perchè utilizzare un modello?

- Algo160 usa il linguaggio naturale, tuttavia in questo modo la semantica diventa incomprensibile. La scelta più adatta è scrivere un modello con una funzione che spiega, per esempio, che cosa sono i conflitti.

- Nello *standard* del C++ la descrizione richiama un modello. La scelta migliore è quella di scriverlo in modo formale, per rendere più chiare le cose.

Semantica Operazionale: da un significato operazionale, descrive ogni operazione del programma nel dettaglio. Si divide in due tipi:

- Semantica Naturale;
- Semantica Operazionale Strutturale: è un interprete, si occupa di dare all'interpretatore le istruzioni senza alcuna ambiguità.

Semantica Denotazionale: in questo caso si parla di funzione da stato a stato, questo però non significa sempre che la funzione termina sempre, può accadere che la funzione non sia definita. Può descrivere la terminazione con i possibili stati raggiunti dalle variabili. In questo caso i tipi sono:

- Stile Semantico Diretto
- Stile Semantico di Continuazione: con *jump* e chiamate di funzione.

Semantica Assiomatica è la semantica degli invarianti. Specifica il significato di un programma rispetto a pre e postcondizioni.
Se so per certo che il programma termina, e si ha la prova di terminazione, si parla di Correttezza Totale, altrimenti ci dobbiamo limitare alla Correttazza Parziale.

Quale sematica scegliere? Dipende dall'obiettivo. Esistono diversi criteri di selezione della sematica: il costrutto del linguaggio (imperativo, funzionale, concorrente/parallelo ..), per che cosa si usa la semantica (per creare un prototipo, analisi di programma, verifica di un programma, comprendere un linguaggio, ..).

Durante il corso faremo uso del *While language*

```
S ::= x :=a | skip | S1;S2
            | if b then S1 else S2
            | while b do S
            | repeat S until b
```

e della notazione BNF che permette di scrivere in modo compatto una grammatica.

Categorie Sintattiche con While language

- numeri
  n ∈ Num
- variabili x ∈ Var
  Useremo il tipo numerico intero e nei valori letterali non specificheremo la sintassi.
- arithmetic expression a ∈ Aexp `a ::= n | x | a1+a2 | a1*a2 | a1-a2`
  Non c'è la divisione per 0, perchè provoca un errore *run-time* che inificia la semantica e non la sintassi. Questo però deve essere presente durante l'analisi statica perchè è proprio il fenomeno che vogliamo evitare.

- boolean a ∈ Bexp `b ::= true | false | a1=a2`
  `| a1<=a2 |` ¬ `b |` `b1` ∨ `b2`

- statements S ∈ Stm `S ::= x :=a | skip | S1;S2`
  `| if b then S1 else S2`
  `| while b do S`

Categorie Semantiche

- Modello per il tipo intero N = {..-2,-1,0,1,2,..} ∈ Z
  T = {tt, ff} semantica dei valori di verità
  La memoria memorizza lo stato del calcolo, dunque le variabili sono locazioni di memoria.
  State = Var → N (variabili che occorrono in un programma)
  lookup sx
  upadate $s'x$ = s[y → v]
  $s'x = \begin{cases} sx & \texttt{x} \neq \texttt{y} \\ v & \texttt{if x = y} \end{cases}$