Cenni informali di Calcolabilità

Corso di Linguaggi di Programmazione

Polo Didattico e di Ricerca di Crema

Uno dei grandi obbiettivi dei linguaggi di programmazione ad alto livello è stato lo sviluppo di costrutti linguistici che—se usati opportunamente—riducono la probabilità di introdurre errori nei programmi, e al tempo stesso supportano metodologie strutturate di testing e debugging. Il ruolo di tali costrutti sarebbe meno strategico ed essenziale se disponessimo di strumenti per verificare automaticamente la correttezza dei programmi. Purtroppo, possiamo dimostrare formalmente che ciò è impossibile. Queste note si propongono di illustrare informalmente come si giunge a simili risultati negativi.

1 Dell'indecidibilità della terminazione e di altre proprietà

Tra le proprietà di correttezza dei programmi, la terminazione è una delle più naturali. Dato un programma, possiamo verificare automaticamente che non cada in un ciclo infinito? Purtroppo dimostreremo che la risposta è *no*.

Per le nostre dimostrazioni informali useremo il linguaggio C e due funzioni di sistema appartenenti ai sistemi operativi della famiglia UNIX. In particolare:

- La chiamata di sistema `exit`(n) fa terminare istantaneamente il programma che la esegue, restituendo un "valore di uscita" n. Una `exit(0)` viene sempre implicitamente posta al termine della procedura `main` di ogni programma.
- La chiamata di sistema `system`(c), dove c è una stringa contenente un comando `<prog name> <arg1>...<argn>`, equivale in tutto e per tutto ad eseguire il comando c dalla command line. Se il comando c termina, la `system` restituisce il valore di uscita n specificato nella `exit` eseguita da c.

Ora supponete che vi sia dato un programma `term` e che vi sia detto che esso verifica la terminazione di un qualunque programma, cioè:

> *Semantica di term*: `term <prog name>` termina con valore di uscita 1 se il programma `<prog name>` termina, altrimenti (se il programma va in ciclo) `term <prog name>` termina con valore di uscita 0.

Come vedete, `term` è poco ambizioso e si propone di verificare solo programmi senza argomenti. Potremmo addirittura restringerci a programmi che non eseguono I/O. Ciononostante, il vostro compito è di dimostrare che quanto vi è stato detto non può essere vero, cioè che la semantica di `term` non può essere quella specificata qui sopra. A questo scopo scrivete il seguente programma, che memorizzerete col nome di `patologico`:[1]

```
main() {
     if( system("term patologico") == 0 ) exit(0);
     else while( 1 ) {}
}
```

Ora supponete di eseguire dalla command line il comando:

```
> term patologico
```

e domandatevi come si comporta. Esistono 3 possibilità:

1. `term patologico` termina con valore di uscita 0. Di conseguenza la funzione `system("term patologico")` restituisce 0 e la condizione nell'`if` di `patologico` risulta vera. Quindi `patologico` termina, eseguendo una `exit`(0). Ma allora `term` non rispetta la semantica promessa (secondo la quale dovrebbe restituire 1).

2. `term patologico` termina con valore di uscita 1. In questo caso la funzione `system("term patologico")` restituisce 1 e la condizione nell'`if` di `patologico` risulta falsa. Quindi `patologico` entra nel ciclo `while` infinito e non termina. Ma allora `term` non rispetta la semantica promessa (secondo la quale dovrebbe restituire 0).

3. `term patologico` non temina, oppure termina con valore di uscita diverso da 0 e 1. Anche in questo caso, `term` non rispetta la semantica data.

Questo dimostra che in ogni caso il programma `term` non può comportarsi come vi è stato detto. Poichè questo vale per qualunque programma `term` dato, concludiamo che *non è possibile scrivere un programma che verifichi se un qualunque altro programma termina o meno.*

Da questo risultato negativo si possono facilmente derivare risultati analoghi per gli altri controlli che ci piacerebbe eseguire automaticamente. Ad esempio supponiamo di voler controllare che gli indici dei vettori rimangano nel giusto range (in C, tra 0 e la dimensione del vettore meno uno). Come prima, supponiamo che vi venga detto che esiste un comando `check_index <prog> <input>` che esegue proprio quel controllo sul programma dato, applicato all'input specificato, e che restituisce 1 se qualche indice finisce fuori dei limiti, altrimenti 0. Voi, per dimostrare che `check_index` non può comportarsi così, scrivete i seguenti programmi, che memorizzerete coi nomi `P` e `Q`, rispettivamente:[2]

[1] La formulazione originale di `patologico` è stata semplificata come segue grazie ad un suggerimento di Sandro Bosio.

[2] Ricordiamo che `argv[1]` coincide con il primo argomento passato al programma dalla command line. Per semplicità, inoltre, supponiamo di avere un operatore '+' che concatena due stringhe.

```
main(argc, argv) {       // prog. P
     int v[10];
     system(argv[1]);
     v[11]=3;            // indice fuori dei limiti
}

main(argc, argv) {       // prog. Q
     exit( system("check_index P "+ argv[1]) );
}
```

Ora `check_index P X` dovrebbe restituire 1 se e solo se `v[11]=3` viene eseguita, e cioè se e solo se la funzione `system("X")` chiamata subito prima termina. Ma allora il programma `Q` qui sopra sarebbe proprio il programma `term` che abbiamo già dimostrato non poter esistere! Ne segue che `check_index` non può comportarsi come vi era stato detto.

Il vantaggio di questo tipo di dimostrazione è che si estende immediatamente ad altri controlli. Basta sostituire l'istruzione `v[11]=3` in `P` con una istruzione che genera il tipo di errore che si vuole verificare (ad es., si veda l'esercizio 1).

Riassumendo, *è impossibile verificare esattamente qualunque proprietà di un programma che dipenda dall'esito della computazione.*

Naturalmente, si possono eseguire controlli automatici imperfetti. Ad esempio, molti compilatori segnalano segmenti di codice irraggiungibili, e dunque inutili; tuttavia, i risultati teorici negativi ci dicono con certezza che questi controlli non saranno in grado di scoprire *tutti* i segmenti irraggiungibili di *qualunque* programma.

Se il compilatore trova una chiamata di funzione dove i tipi dei parametri attuali non corrispondono a quelli dei parametri formali, segnala un errore. Ma è possibile che quella chiamata non venga mai eseguita, dunque il programma funzionerebbe bene nonostante quell'errore sintattico. In questo caso viene rifiutato un programma dal comportamento corretto.

Alcuni controlli risultano possibili a run-time. Molti compilatori introducono controlli sugli indici dei vettori ovunque ci sia un riferimento ad un loro elemento. Tuttavia, in un certo senso, i controlli a run-time avvengono troppo tardi; servono ad individuare esattamente il punto in cui si verifica l'errore, e non ad evitare a priori malfunzionamenti.

2 Perchè serve un modello formale

Le "dimostrazioni" utilizzate nel capitolo precedente hanno il pregio di essere facili da capire, ma hanno dei gravi limiti:

- Non sono *formali*, dunque non danno garanzie sulla correttezza delle conclusioni. In particolare, ci siamo affidati ad una descrizione incompleta e meramente intuitiva del comportamento delle funzioni di sistema UNIX `system` ed `exit`, e dei costrutti C `if`, `==` e `while`.

- Non sono dimostrazioni *generali*. Infatti sappiamo che il comportamento delle funzioni di sistema di UNIX può cambiare da una versione a un'altra. Similmente, il comportamento dei costrutti C può cambiare a seconda di quale compilatore viene usato e—peggio ancora—di quale processore li esegue. Pertanto dovremmo domandarci se l'indecidibilità della terminazione continuerebbe a valere se cambiassimo sistema operativo, processore, o addirittura *linguaggio* di programmazione.

La potenziale dipendenza dei risultati negativi dalla scelta del linguaggio ci porta spontaneamente ad un'altra questione:

> **Quesito 1:** *dati due linguaggi di programmazione L_1 ed L_2, è vero o no che per ogni programma P_1 in L_1 esiste un programma P_2 in L_2 che calcola le stesse cose?*

La rilevanza del quesito rispetto alla precedente domanda dovrebbe essere chiara: se la risposta fosse "*sì*", allora ciò che non è possibile usando L_2 non sarebbe possibile nemmeno con L_1, e qualunque risultato negativo per L_2 sarebbe ereditato da L_1.

La risposta al Quesito 1 è rilevante anche per un altro aspetto dei linguaggi di programmazione: vorremmo che essi fossero *general purpose*, cioè in grado di calcolare tutto ciò che è possibile calcolare (qualunque cosa voglia dire). Ovviamente, se per qualche coppia di linguaggi L_1 ed L_2 e per qualche programma P_1 la risposta al quesito fosse "*no*", ne seguirebbe che L_2 non è general purpose.

Infine, considerate il problema di identificare "tutto ciò che è possibile calcolare" (si veda il paragrafo precedente). Cosa può significare questa espressione?

Le *macchine di Turing* (MdT) sono il formalismo comunemente usato per affrontare le domande espresse qui sopra, e per superare i limiti delle descrizioni informali già citati. Ecco le ragioni di questa scelta:

- Le MdT sono oggetti matematici, la cui definizione completa è *formale* e *molto compatta*. Questo permette di costruire prove rigorose e di dimensioni maneggevoli, con tutte le garanzie che ne seguono. Pensate a cosa significherebbe definire rigorosamente, in termini matematici, il comportamento di un sistema operativo o di un compilatore C—di cui ci servirebbe una minima parte, che però sarebbe difficile da isolare da tutto il resto. Le MdT ci permettono di "distillare" gli aspetti importanti e concentrarci su quelli.

- Grazie alla definizione formale e all'assenza di dettagli inutili, i programmi delle MdT hanno un comportamento non ambiguo, che non dipende da specifiche versioni di sistema operativo e compilatore (che non vengono modellati). Il processore è unico e il suo comportamento definito matematicamente, come i programmi.

- Nonostante la loro semplicità, le MdT possono calcolare tutto ciò che può essere calcolato con gli altri modelli e strumenti di calcolo a noi noti, compreso tutto quanto è calcolabile con i linguaggi di programmazione

oggi esistenti. In altre parole, tutto ciò che oggi noi sappiamo calcolare, si può esprimere con una MdT. In questo senso oggi consideriamo come general purpose i linguaggi di programmazione *Turing equivalenti*, cioè quei linguaggi che hanno le stesse potenzialità di calcolo delle macchine di Turing.[3]

- L'estrema semplicità delle MdT facilita la dimostrazione della generalità dei linguaggi di programmazione. Da un lato, con un linguaggio ad alto livello è facile simulare una qualsiasi macchina di Turing. Dall'altro, dimostrare che il linguaggio ad alto livello non è più potente delle MdT è abbastanza semplice, in quanto i linguaggi macchina su cui viene tradotto possono essere interpretati da opportune MdT senza troppe difficoltà.

3 Le Macchine di Turing

Una macchina di Turing è una n-upla $M = \langle V, S, R, s_0 \rangle$, dove:

- V è un insieme di *simboli*. Essi rappresentano i possibili valori che possono essere memorizzati nelle locazioni della memoria di M. Per usare un esempio classico, se le parole di memoria fossero singoli byte, allora potremmo usare come V l'insieme degli interi compresi tra 0 e 255, ma qualunque altro insieme di 256 elementi andrebbe altrettanto bene. Uno dei membri di V ha un ruolo particolare (è una sorta di *blank*), ed è denotato da $\flat$.

- S è un insieme di *stati*. Questi hanno lo stesso ruolo delle *labels* nei linguaggi di programmazione: identificano una istruzione del programma della macchina.

- s_0 è lo *stato iniziale*, corrispondente alla prima istruzione eseguita dalla macchina.

- R è il programma della macchina. Consiste di un insieme finito di n-uple

$$\langle s, v, v', m, s' \rangle ,$$

  dove $s, s' \in S$, $v, v' \in V$ e $m \in \{\mathbf{sx}, \mathbf{dx}\}$.

Per capire come interpretare queste strane istruzioni dobbiamo riferirci allo schema di fig. **??**. La memoria della MdT viene rappresentata come un nastro illimitato nelle due direzioni. In ciascuna delle "celle" del nastro è memorizzato un simbolo di V. Le caselle inizialmente contengono tutti $\flat$, fatta eccezione per un numero finito di locazioni (questo stato iniziale del nastro rappresenta l'*input* di M). La MdT dispone di una testina che può leggere e scrivere il valore della cella che le si trova sotto. Ad ogni passo, il nastro può essere spostato di una posizione a destra o a sinistra, di modo che la testina è potenzialmente in grado

[3] Chiaramente, questo non significa che il numero di passi richiesti da un dato programma scritto in un linguaggio di alto livello o la sua dimensione siano paragonabili a quelli della MdT corrispondente.

di leggere e scrivere una qualunque locazione del nastro (dopo un adeguato numero di passi). Ora, l'insieme di tutte le n-uple

$$\begin{gathered}\langle s, v_1, v'_1, m_1, s'_1\rangle \\ \langle s, v_2, v'_2, m_2, s'_2\rangle \\ \dots \\ \langle s, v_n, v'_n, m_n, s'_n\rangle\end{gathered}$$

in R, con lo stesso valore di s, costituisce l'analogo del seguente frammento di pseudo-codice (basato sul costrutto `switch` del C):

```
s:   v = <simbolo sotto la testina>
     switch(v) {
          case v1: <scrivi v1' sotto la testina>;
                   if(m1==sx) <sposta la testina a sinistra>
                   else       <sposta la testina a destra>;
                   goto s1';
          case v2: <scrivi v2' sotto la testina>;
                   if(m2==sx) <sposta la testina a sinistra>
                   else       <sposta la testina a destra>;
                   goto s2';
          ....
          case vn: <scrivi vn' sotto la testina>;
                   if(mn==sx) <sposta la testina a sinistra>
                   else       <sposta la testina a destra>;
                   goto sn';
      }
```

Formalmente, le computazioni delle MdT sono modellate da sequenze di *configurazioni*, che sono a loro volta sequenze infinite della forma:

$$\dots \flat\, \flat\, v_1 v_2 \dots v_{k-1}\, s\, v_k v_{k+1} \dots v_f \flat\, \flat \dots$$

in cui solo un numero finito di elementi è diverso da $\flat$, e dove s segnala sia lo stato della macchina in quella configurazione, sia la posizione della testina (si conviene che il simbolo sotto la testina sia quello a destra di s, ossia v_k). I simboli $\flat$ e $v_1 \dots v_f$ modellano il contenuto del nastro in quella configurazione. Si noti che ogni configurazione può essere rappresentata in modo finito e senza ambiguità tramite la sottosequenza compresa tra i simboli non-$\flat$ estremi (nell'esempio precedente, se $v_1 \neq \flat$ e $v_f \neq \flat$, allora la rappresentazione finita è $v_1 v_2 \dots s v_k v_{k+1} \dots v_f$). Definiamo ora la relazione di transizione tra configurazioni (denotata con $\rightsquigarrow$) per M; nel seguito α e β denotano sequenze di simboli di V. Adottando la rappresentazione finita, possiamo assumere che α e β siano finite.

$$\begin{array}{lll} \alpha s v \beta \rightsquigarrow \alpha v' s' \beta & \text{se} & \langle s, v, v', \mathbf{dx}, s'\rangle \in R \\ \alpha v_1 s v \beta \rightsquigarrow \alpha s' v_1 v' \beta & \text{se} & \langle s, v, v', \mathbf{sx}, s'\rangle \in R. \end{array}$$

Possiamo ora definire formalmente cos'è una *computazione*, e cioè una sequenza di configurazioni $\chi_1\chi_2\ldots\chi_i\ldots$, tale che per ogni $i > 0$, $\chi_i \rightsquigarrow \chi_{i+1}$. Notate come poche righe di definizioni formali (riassunte qui sotto) definiscano cos'è una MdT e come si comporta.

- Una MdT è una n-upla $\langle V, S, R, s_0\rangle$, dove $R \subseteq S \times V \times V \times \{\mathbf{sx}, \mathbf{dx}\} \times S$ and $s_0 \in V$.
- Una configurazione di una MdT è una sequenza della forma

  $$\ldots \flat\, \flat\, v_1 v_2 \ldots v_{k-1}\, s\, v_k v_{k+1} \ldots v_f \flat\, \flat \ldots$$

  dove $s \in S$ e $v_i \in V$ $(i = 1 \ldots f)$.
- La relazione di transizione è specificata da:

  $$\begin{array}{lll} \alpha s v \beta \rightsquigarrow \alpha v' s' \beta & \text{se} & \langle s, v, v', \mathbf{dx}, s'\rangle \in R \\ \alpha v_1 s v \beta \rightsquigarrow \alpha s' v_1 v' \beta & \text{se} & \langle s, v, v', \mathbf{sx}, s'\rangle \in R\,. \end{array}$$

- Una computazione è una sequenza di configurazioni $\chi_1\chi_2\ldots\chi_i\ldots$, tale che per ogni $i > 0$, $\chi_i \rightsquigarrow \chi_{i+1}$.

3.1 Cenni sulla dimostrazione formale di indecidibilità

Non diamo qui una dimostrazione formale dell'impossibiltà di decidere la terminazione di una MdT. Accenniamo invece a come gli elementi della dimostrazione informale trovano riscontro in quella formale.

Nella dimostrazione informale è stato importante l'uso di file names per riferirsi ai programmi. Per etichettare con dei "nomi" le macchine di Turing ci occorrono alcuni preliminari matematici.

È noto che i numeri naturali sono isomorfi alle coppie di naturali, e dunque la cardinalità di $\mathbb{N}$ coincide con quella di $\mathbb{N}^2$. Per dimostrare questa corrispondenza si organizzano le coppie in una matrice:

$$\begin{array}{cccc} (0,0) & (0,1) & (0,2) & \ldots \\ (1,0) & (1,1) & (1,2) & \ldots \\ (2,0) & (2,1) & (2,2) & \ldots \\ \vdots & & & \end{array}$$

e "visitando" questa matrice lungo le diagonali

$$\begin{array}{ccc} (0,0) & & \\ (1,0) & (0,1) & \\ (2,0) & (1,1) & (0,2) \\ \vdots & & \end{array}$$

si enumerano tutte le coppie in $\mathbb{N}^2$, ottenendo la seguente corrispondenza biunivoca tra $\mathbb{N}$ ed $\mathbb{N}^2$:

$$\begin{array}{lll} 0 \leftrightarrow (0,0) & & \\ 1 \leftrightarrow (1,0) & 2 \leftrightarrow (0,1) & \\ 3 \leftrightarrow (2,0) & 4 \leftrightarrow (1,1) & 5 \leftrightarrow (0,2) \\ \vdots & & \end{array}$$

Lo stesso procedimento si può ripetere per mettere in corrispondenza le triple di naturali con i singoli naturali. Data una tripla (x, y, z), prima si ricava il numero y' corrispondente a (y, z), e poi si ottiene il numero x' corrispondente a (x, y') che codifica la tripla.[4]

Chiaramente, riapplicando lo stesso metodo, possiamo ottenere corrispondenze biunivoce tra $\mathbb{N}$ e $\mathbb{N}^k$, per ogni naturale k. Sfruttando tali corrispondenze, e applicando la codifica "diagonale" alla matrice:

$$\begin{array}{llll} (0) & (1) & (2) & \dots \\ (0,0) & (0,1) & (0,2) & \dots \\ (0,0,0) & (0,0,1) & \dots & \\ (0,0,0,0) & \dots & & \\ \vdots & & & \end{array}$$

(dove la riga k-esima enumera $\mathbb{N}^k$) concludiamo che l'insieme di tutte le stringhe finite di naturali è isomorfo all'insieme dei naturali stessi. Continuando su questa strada, riusciamo a codificare ogni struttura finita $\langle V, S, R, s_0 \rangle$ con un naturale e così possiamo *identificare le MdT con i numeri naturali*; questi possono giocare il ruolo di file name per le MdT.

La `system`, invece, corrisponde alla *MdT Universale*, che dato un intero che codifica una coppia (MdT,nastro iniziale) simula la computazione della MdT data a partire dal nastro specificato.

Invece della `exit` abbiamo stati finali diversi, che corrispondono ai diversi valori di uscita del programma.

Infine, invece di costruire il programma `patologico` assemblando costrutti C, costruiamo una macchina di Turing da altre più semplici, ad esempio identificando gli stati terminali dell'una con quelli iniziali dell'altra per ottenere sequenze di comandi.

4 Esercizi

1. Dimostrare, nello stile informale utilizzato per il controllo degli indici degli array, che non possono esistere programmi che controllino esattamente:
   (a) l'uso corretto delle `union` in C;
   (b) la presenza di codice irraggiungibile.

[4] In altre parole è come se codificassimo (x, y, z) come $(x, (y, z))$.

2. Definire una macchina di Turing che sommi due cifre binarie.

3. Scrivere un simulatore di MdT in un linguaggio di alto livello per ogni paradigma (ad es., Java, ML, Prolog).