Note di semantica assiomatica

Appunti ad uso degli studenti di Programmazione (I modulo)
Corso di Laurea in Informatica
Università degli Studi di Pisa

Anno accademico 1997/98

Paolo Baldan e Paolo Mancarella

Indice

1 Introduzione

La diffusione dei sistemi informatici cresce in modo sempre più rapido e inarrestabile e parallelamente aumenta la loro importanza. Prenotazioni aeree o ferroviarie, operazioni bancarie, sistemi di segnalazione per automobili e treni, sistemi di gestione dei bagagli negli aereoporti, strumentazione degli aereoplani, traffico telefonico e telematico sono ormai completamente gestiti tramite sistemi controllati da calcolatori. Per i clienti delle agenzie di viaggio, o i passeggeri di treni ed areoplani, il corretto funzionamento di questi sistemi è di fondamentale importanza. In alcuni casi un errore di funzionamento può semplicemente creare delle situazioni spiacevoli, ma in altri può avere effetti disastrosi. Dunque i programmi eseguiti su calcolatori che controllano processi di questo tipo devono funzionare correttamente, o in altri termini, devono soddisfare le specifiche in base alle quali sono stati costruiti. Uno dei compiti dell'informatica è dunque l'individuazione di tecniche per la *verifica della correttezza* dei programmi e per lo *sviluppo di programmi corretti*.

I *linguaggi di programmazione* sono linguaggi "artificiali" che permettono di esprimere *algoritmi*, ovvero procedimenti risolutivi di problemi, in modo che un elaboratore automatico sia poi in grado di eseguirli. Un *programma* non è altro che la traduzione di un algoritmo in un particolare linguaggio di programmazione.

La definizione di un linguaggio (artificiale e non) prevede essenzialmente due aspetti: una descrizione della *sintassi* del linguaggio, ovvero delle frasi legali del linguaggio, e una descrizione della *semantica* del linguaggio, ovvero del significato delle frasi sintatticamente corrette. Nel caso dei linguaggi di programmazione, la sintassi descrive la struttura di un programma, cioè spiega in che modo frasi e simboli di base possono essere composti al fine di ottenere un programma legale nel linguaggio. A differenza dei linguaggi naturali, la sintassi dei linguaggi di programmazione può essere descritta in modo rigoroso utilizzando i formalismi messi a disposizione dalla teoria dei linguaggi formali. Non è scopo di queste note affrontare lo studio di tale teoria. Nel seguito faremo riferimento semplicemente ai pochi concetti di base sulla sintassi dei linguaggi già noti al lettore.

Nella storia, peraltro recente, dei linguaggi di programmazione vi è un evidente contrasto tra come ne viene descritta la sintassi e come ne viene descritta la semantica. Di solito, a una descrizione rigorosa e formale della sintassi, è affiancata una descrizione poco rigorosa e del tutto informale della semantica: i manuali di riferimento dei linguaggi di programmazione più comuni testimoniano questa situazione.

L'approccio più comune alla verifica dei programmi è dunque stato, per lungo tempo, il *ragionamento operazionale* (ovvero l'analisi delle possibili sequenze di esecuzione del programma dato) fondato su una comprensione informale della semantica del linguaggio di programmazione. L'informalità dell'approccio, se da un lato lo rende accessibile ad un vasto pubblico di non esperti, dall'altro ne rappresenta il limite fondamentale. Infatti, per programmi complessi o concettualmente intricati è complicato seguire tutte le possibili sequenze di esecuzione. Inoltre, la dimostrazione di correttezza poggia su considerazioni informali e quindi è, in qualche modo "soggettiva", difficilmente comunicabile, soggetta ad errori difficilmente riscontrabili.

Per poter specificare in modo rigoroso le proprietà che vogliamo che un programma soddisfi ed essere in grado di verificare se tali proprietà sono o meno soddisfatte occorre dare una definizione formale della semantica del linguaggio di programmazione e del linguaggio di specifica delle proprietà dei programmi.

In queste note presentiamo il cosiddetto approccio *assiomatico* alla semantica dei linguaggi di programmazione. Il linguaggio che permette di esprimere, o con termine tecnicamente più corretto di *specificare*, le proprietà di interesse del programma è il linguaggio della *logica dei predicati*. La logica ci fornisce anche il concetto di *proof system*, un insieme di assiomi e regole che permetteranno di provare formalmente che un dato programma soddisfa le proprietà desiderate. La dimostrazione procederà in modo guidato dalla sintassi, per induzione sulla struttura del programma. Il linguaggio di programmazione considerato è un linguaggio *imperativo*, con comando vuoto, comando di assegnamento ed i costrutti di sequenza, condizionale e iterativo. I tipi di dato (impliciti) del linguaggio sono booleani, interi e sequenze (array) di interi.

L'origine di questo approcco alla verifica dei programmi è riconducibile a Turing [1949], ma il primo sforzo costruttivo è attribuibile a Floyd [1967] e Hoare [1969]. Floyd propose un insieme di assiomi e regole per la verifica di diagrammi di flusso. Successivamente, Hoare modificò ed estese il sistema di Floyd allo scopo di trattare un linguaggio imperativo di tipo *while* con procedure. Originariamente l'approccio fu introdotto, non solo per provare e verificare proprietà dei programmi, ma anche come metodo per la spiegazione del significato dei costrutti di programmazione. La semantica di un costrutto era definita indicando gli assiomi e le regole che permettevano di derivare proprietà del costrutto stesso. Per questa ragione l'approccio è tradizionalmente chiamato *assiomatico*. Dopo il lavoro di Hoare, sono stati proposti innumerevoli altri sistemi di prova alla Hoare, per la verifica di linguaggi contenenti i più disparati costrutti programmativi, fino ai linguaggi per la programmazione parallela e distribuita.

Nel lavoro di Hoare, le regole di prova permettono una verifica a posteriori di programmi esistenti, ma non uno sviluppo sistematico dei programmi stessi. Il tentativo di superare questa limitazione ha determinato la nascita del filone di ricerca noto come *sviluppo sistematico di programmi*, iniziato da Dijkstra [1976]. È interessante notare come le regole del sistema di prova per la verifica a posteriori rimangano utili e addirittura rappresentino le linee guida per la formulazione di strategie per lo sviluppo sistematico di programmi.

Come nota conclusiva, è naturale chiedersi se la verifica dei programmi possa essere completamente automatizzata. È affascinante l'idea di poter dare in input ad un calcolatore un programma e la sua specifica e semplicemente attendere una risposta. Sfortunatamente la teoria della computabilità stronca sul nascere questo sogno: la verifica automatica delle proprietà dei programmi è in generale (a meno di restringerci a particolari classi di proprietà o di programmi) non decidibile e quindi impossibile da implementare.

L'organizzazione di queste note è la seguente. Il Paragrafo 2 contiene i preliminari matematici e fissa alcune convenzioni notazionali adottate in queste note. Il Paragrafo 3 presenta il linguaggio di programazione che verrà studiato. In primo luogo si introduce il linguaggio delle espressioni (booleane e intere) e la loro semantica. Quindi viene definito il linguaggio dei comandi, un linguaggio imperativo con comando vuoto, comando di assegnamento ed i costrutti di sequenza, condizionale e iterativo, e viene data una semantica operazionale informale di tale linguaggio. Il Paragrafo 4 introduce la semantica assiomatica del linguaggio dei comandi basata su Triple di Hoare. Il Paragrafo 5 introduce l'idea di annotazione di programmi con asserzioni e dà alcune indicazioni riguardo a come e quanto un programma dovrebbe essere annotato. Nel Paragrafo 6 si discute la nozione di variabile di specifica ed il suo ruolo nella formalizzazione di alcuni problemi di programmazione. Il Paragrafo 7 presenta alcuni "schemi di programma" che, istanziati, si prestano alla soluzione di molti problemi comuni, e ne prova la correttezza. Il Paragrafo 8 propone alcune euristiche che permettono di derivare le asserzioni corrette da inserire come invarianti nei comandi di tipo iterativo che risolvono una data specifica. Infine il Paragrafo 9 contiene una raccolta di esercizi risolti che illustrano le tecniche presentate in queste note.

Prima di iniziare tentiamo di chiarire con un semplice esempio quali sono le motivazioni e gli obiettivi dell'approccio assiomatico.

Perché usare la logica? Perché provare la correttezza dei programmi? Un esempio

Supponiamo di aver appena finito di scrivere un programma di 30000 righe. Tra le altre cose, il programma dovrebbe calcolare, come risultato intermedio, il quoziente q ed il resto r della divisione del numero intero non negativo x per l'intero positivo y. Per esempio, con $x = 7$ e $y = 2$, il programma calcola $q = 3$ e $r = 1$.

Il nostro programma è il seguente:

$$
\begin{array}{l}
\ldots \\
q, r := 0, x; \\
\textbf{while } r > y \textbf{ do} \\
\qquad q, r := q + 1, r - y \\
\textbf{endw} \\
\ldots
\end{array}
$$

dove i puntini "…" rappresentano le parti del programma che precedono e seguono il calcolo del quoziente (e del resto). L'algoritmo è molto semplice: si sottrae ripetutamente il divisore y da una copia di x, mantenendo traccia di quante sottrazioni sono state effettuate, finché un'altra sottrazione darebbe come risultato un intero negativo.

Siamo quindi pronti a verificare il funzionamento del programma. È chiaro fin da subito che inizialmente il divisore dovrebbe essere maggiore di zero e che dopo l'esecuzione del ciclo le variabili dovrebbero soddisfare la formula:

$$x = y * q + r,$$

così, per controllare l'avanzamento del calcolo, inseriamo delle operazioni di output, ottenendo:

```
...
write('Dividendo x =', x);
write('Divisore y =', y);
q, r := 0, x;
while r > y do
        q, r := q + 1, r − y
        write('y*q+r = ', y ∗ q + r);
endw
...
```

Purtroppo, in questo modo, otteniamo un output estremamente voluminoso, dato che il frammento di programma considerato include un ciclo. L'output è dunque quasi incomprensibile ed inutilizzabile. Dovremmo scegliere più attentamente ciò che forniamo in output.

Supponiamo che il compilatore sia stato esteso in modo da fornire la seguente possibilità: se una espressione booleana racchiusa tra parentesi graffe $\{B\}$ compare in un certo punto del programma, allora, ogni qualvolta, durante l'esecuzione, il "flusso del controllo" raggiunge questo punto, l'espressione B è valutata. Se risulta falsa, allora il valore delle variabili di programma che compaiono in B viene stampato, altrimenti l'esecuzione continua normalmente. Queste espressioni booleane sono dette *asserzioni*, poiché, in effetti, asseriscono una proprietà che dovrebbe essere vera ogni qualvolta il "flusso del controllo" le raggiunge. Tali asserzioni all'interno di un programma ne facilitano anche la comprensione in quanto specificano quale dovrebbe essere, secondo il programmatore, lo stato delle variabili durante l'esecuzione.

Utilizzando questo nuovo ipotetico strumento, otteniamo il programma, arricchito con asserzioni:

```
    ...
{y > 0}
    q, r := 0, x;
    while r > y do
            q, r := q + 1, r − y
    endw
{x = y ∗ q + r}
    ...
```

Controllare i risultati è ora più semplice, vista la minore dimensione dell'output. Il sistema di verifica delle asserzioni determina un errore durante un'esecuzione trovando che il valore di y è 0 immediatamente prima del ciclo per il calcolo del quoziente, ed in sole quattro ore riusciamo a trovare l'errore nel calcolo di y e a risolverlo.

Ma poi, impieghiamo un giorno nel tentativo di scoprire la ragione di un errore per il quale non ci è stato fornito nessun messaggio dal sistema di verifica delle asserzioni. Alla fine del calcolo del quoziente otteniamo che

$$x = 6,\ y = 3,\ q = 1,\ r = 3.$$

Il problema è che il resto r dovrebbe essere minore del divisore y, ed invece non lo è. Dopo un momento realizziamo che la condizione del ciclo dovrebbe essere $r \geq y$ anziché $r > y$. Se l'asserzione che segue il ciclo fosse stata sufficientemente forte, ovvero se avessimo usato l'asserzione $\{x = y * q + r \ \wedge \ r < y\}$ anziché $\{x = y * q + r\}$ avremmo risparmiato un giorno di lavoro! Perché non ci abbiamo pensato?

Risolviamo dunque l'errore e inseriamo l'asserzione più forte:

$$
\begin{array}{l}
\quad \ldots \\
\{y \geq 0\} \\
\quad q, r := 0, x; \\
\quad \textbf{while } r \geq y \textbf{ do} \\
\qquad\qquad q, r := q + 1, r - y \\
\quad \textbf{endw} \\
\{x = y * q + r \ \wedge \ r < y\} \\
\quad \ldots
\end{array}
$$

Le cose sembrano funzionare per un po', ma un giorno otteniamo un output incomprensibile. Il calcolo del quoziente fornisce un resto negativo $r = -2 \ldots$ ma il resto *non* dovrebbe mai essere negativo! Dopo un po' scopriamo che r risultava negativo perché inizialmente il valore di x era -2. Ahhh, un altro errore nel calcolo dell'input del sottoprogramma per il calcolo del quoziente: il valore di x doveva essere non negativo! Ma avremmo potuto trovare l'errore prima e risparmiare altri due giorni di lavoro se solo avessimo inserito delle asserzioni iniziali e finali sufficientemente forti. Ancora una volta, dunque, risolviamo l'errore e rafforziamo le asserzioni nel modo opportuno:

$$
\begin{array}{l}
\quad \ldots \\
\{x \geq 0 \ \wedge \ y > 0\} \\
\quad q, r := 0, x; \\
\quad \textbf{while } r \geq y \textbf{ do} \\
\qquad\qquad q, r := q + 1, r - y \\
\quad \textbf{endw} \\
\{x = y * q + r \ \wedge \ 0 \leq r < y\} \\
\quad \ldots
\end{array}
$$

Sarebbe certamente meglio poter definire le asserzioni corrette in un modo meno *ad hoc*, non operare in modo guidato dagli errori. Sicuramente un problema è stato non dare fin dall'inizio una specifica corretta di cosa il segmento di programma dovesse fare. Avremmo dovuto scrivere l'asserzione iniziale $\{x \geq 0 \ \wedge \ y > 0\}$ e quella finale $\{x = y * q + r \ \wedge \ 0 \leq r < y\}$, che altro non sono che la definizione del quoziente e del resto, *prima* di scrivere il segmento di programma.

Che dire, invece, dell'errore nella condizione del ciclo? Avremmo potuto evitarlo fin dall'inizio? C'è un modo per dimostrare, solo dal programma e dalle asserzioni, che le asserzioni sono vere quando sono raggiunte dal flusso del controllo? Vediamo cosa si può fare in questo senso.

Supponiamo che prima di entrare nel sottoprogramma per il calcolo del quoziente i valori delle variabili x e y soddisfino l'asserzione

$$\{x \geq 0 \ \wedge \ y > 0\}.$$

Quindi, dopo l'assegnamento $r, q := x, 0$, immediatamente prima del ciclo è sicuramente vera l'asserzione

$$\{r \geq 0 \ \wedge \ y > 0 \wedge \ x = y * q + r\} \tag{†}$$

Ora, se la condizione $r \geq y$ del ciclo è soddisfatta e vale l'asserzione (†), è facile rendersi conto, vista la forma dell'assegnamento che compare come corpo del ciclo, che (†) continuerà a valere anche dopo l'esecuzione di tale assegamento. Questo ci permette di concludere che l'asserzione (†) sarà vera prima di ogni iterazione del ciclo. Possiamo dunque inserire tale asserzione nel programma, ottenendo:

$$
\begin{array}{l}
\quad \ldots \\
\{x \geq 0 \ \wedge \ y > 0\} \\
\quad q, r := 0, x;
\end{array}
$$

$$
\begin{array}{l}
\quad \textbf{while } r \geq y \textbf{ do} \\
\qquad\qquad \{r \geq 0 \ \wedge \ y > 0 \ \wedge \ x = y * q + r\} \\
\qquad\qquad q, r := q + 1, r - y \\
\quad \textbf{endw} \\
\{x = y * q + r \ \wedge \ 0 \leq r < y\} \\
\quad \dots
\end{array}
$$

Ora, come possiamo determinare la condizione del ciclo, o data la condizione, come possiamo dimostrare che è corretta? Quando il ciclo termina la condizione è falsa. Quando questo accade noi vogliamo che l'asserzione finale $\{x = y * q + r \ \wedge \ 0 \leq r < y\}$ sia soddisfatta. Ma si osservi che tale asserzione altro non è che $(\dagger) \ \wedge \ r < y$. Visto che $(\dagger)$ è sempre soddisfatta durante il ciclo, la condizione corretta sarà il complementare di $r < y$, ovvero $r \geq y$. Facile, no?

Sembra quindi che, se fossimo stati in grado di scrivere le asserzioni corrette fin dall'inizio e di ragionare correttamente su asserzioni e programmi, non avremmo commesso così tanti errori. Avremmo saputo che il nostro programma era corretto e non avremmo avuto bisogno di verificarne il funzionamento. Dunque, il tempo speso a provare e riprovare il programma, a cercare gli errori e a tentare di risolverli avrebbe potuto essere speso in modi migliori!

Lo scopo di queste note è appunto lo studio dell'approccio basato su asserzioni, detto approccio *assiomatico*, alla verifica di programmi e allo sviluppo di programmi corretti. Moltissimi concetti e nozioni fondamentali di questo approccio sono apparsi, seppure in modo confuso e impreciso, nell'esempio appena analizzato e attendono solo di essere presentati in modo sistematico e formale, dunque ... rimbocchiamoci le maniche.

2 Preliminari Matematici

Lo scopo di questo paragrafo è quello di introdurre brevemente alcuni preliminari matematici e alcune convenzioni notazionali adottate nelle note.

Fondamentale in matematica è il concetto di insieme. Non faremo riferimento esplicito ad una teoria formale degli insiemi, ma utilizzeremo semplicemente la nozione intuitiva di "insieme come collezione di oggetti". Un insieme può essere rappresentato elencando tutti i suoi elementi tra parentesi graffe. Ad esempio l'insieme che contiene come elementi i numeri naturali 1, 3 e 5 sarà denotato da

$$\{1, 3, 5\}.$$

Quando il numero di elementi è molto grande (o addirittura infinito), si può semplicemente iniziare la lista degli elementi e utilizzare dei puntini di sospensione per rappresentare gli elementi non indicati. Ad esempio l'insieme dei numeri pari minori o uguali a 100 può essere denotato come

$$\{0, 2, 4, \ldots, 98, 100\},$$

mentre l'insieme dei numeri pari può essere rappresentato come:

$$\{0, 2, 4, \ldots\}.$$

Infine, talvolta risulta comodo rappresentare un insieme utilizzando una proprietà soddisfatta da tutti e soli i suoi elementi:

$$\{x \mid x \text{ soddisfa la proprietà } P\}.$$

Ad esempio insieme dei numeri naturali pari sarà $\{x \mid x \text{ numero naturale e } x \ mod \ 2 = 0\}$.

Esempio 2.1 Due insiemi che svolgeranno un ruolo fondamentale in queste note, in quanto contengono i valori denotati dalle espressioni del linguaggio considerato, sono l'insieme dei numeri interi e l'insieme dei valori di verità.

L'insieme dei *numeri interi* è denotato da $\mathbb{Z}$. Gli elementi di $\mathbb{Z}$ verranno scritti in grassetto (ad esempio $\mathbf{2}, -\mathbf{3}$, ecc.) per distinguerli dalle loro rappresentazioni sintattiche (cfr. Paragrafo 3.1). Anche le meta-variabili che indicano generici elementi di $\mathbb{Z}$ saranno in grassetto (ad esempio, $\mathbf{n}, \mathbf{m}, \mathbf{n}'$ denoteranno generici numeri interi).

Con $\mathbb{B}$ denotiamo l'insieme dei *valori di verità* $\{\mathbf{tt}, \mathbf{ff}\}$. Anche per $\mathbb{B}$ utilizziamo la convenzione di indicarne un generico elemento con una meta-variabile in grassetto (ad esempio $\mathbf{b}, \mathbf{b}'$, ecc.).

L'*appartenenza* ad un insieme è indicata, al solito, con il simbolo $\in$. Leggeremo $a \in A$ come a è un elemento dell'insieme A. Sugli insiemi utilizzeremo le usuali operazioni di *unione*, *intersezione*, *prodotto*, ecc. Se A e B sono insiemi, allora:

$$\begin{aligned}
&A \cup B = \{c \mid c \in A \text{ oppure } c \in B\},\\
&A \cap B = \{c \mid c \in A \text{ e } c \in B\},\\
&A \times B = \{(a, b) \mid a \in A \text{ e } b \in B\}.
\end{aligned}$$

Ricordiamo che una *relazione* R tra gli elementi di A e quelli di B è un sottoinsieme del prodotto cartesiano $R \subseteq A \times B$. Usualmente per le relazioni binarie si usa la notazione infissa e si scrive $a\,R\,b$ per indicare che $(a, b) \in R$. Una relazione $R \subseteq A \times A$ si dice

- *riflessiva*, se $\forall a \in A.\ a\,R\,a$;
- *simmetrica*, se $\forall a_1, a_2 \in A.\ (a_1 R\, a_2 \Rightarrow a_2 R\, a_1)$;
- *transitiva*, se $\forall a_1, a_2, a_3 \in A.\ (a_1 R\, a_2 \ \wedge\ a_2 R\, a_3 \ \Rightarrow\ a_1 R\, a_3)$.

Una *funzione (parziale)* $f : A \to B$ è una particolare relazione su $A \times B$, tale che dati comunque $b, b' \in B$, se $(a,b),(a,b') \in f$ allora $b = b'$. Dunque, dato $a \in A$, l'elemento $b \in B$ tale che $(a,b) \in f$, se esiste, è unico, ed è indicato con $f(a)$; altrimenti diremo che $f(a)$ è indefinito. L'insieme A è detto *dominio* e B è detto *codominio* della funzione. La funzione f è *totale* se $f(a)$ è definito per ogni $a \in A$.

Sia f una funzione del tipo $f : A \to B$ e siano, rispettivamente, $a \in A$ e $b \in B$. Allora con la notazione $f[^b/_a]$ indichiamo una nuova funzione da $A \to B$ definita come segue:

$$f[^b/_a](x) = \begin{cases} b & \text{se } x = a \\ f(x) & \text{altrimenti.} \end{cases}$$

Dalla precedente definizione seguono facilmente le seguenti proprietà:

Proprietà 2.2 Sia f una funzione del tipo $f : A \to B$ e siano, rispettivamente, $a \in A$ e $b \in B$. Allora:

(i) $(\forall x :\ x \in A \wedge x \neq a \ \Rightarrow\ (f[^b/_a](x) = f(x)))$

(ii) $(\forall x :\ x \in A \wedge x = a \ \Rightarrow\ (f[^b/_a](x) = b))$

Come già accennato nell'introduzione, il linguaggio formale utilizzato per esprimere le asserzioni, ovvero le proprietà di interesse dei programmi è quello della *logica del prim'ordine*. Verrà assunta una conoscenza di base dell'argomento.[1]

Le dimostrazioni matematiche sono spesso catene di uguaglianze tra espressioni. Lo stile che adotteremo per presentare tali catene, dovuto a Scholten [1900], è il seguente:

$$\begin{array}{ll}
 & ass_1 \\
\equiv & \quad \{ \text{ ragione per cui } ass_1 \equiv ass_2 \ \} \\
 & ass_2 \\
 & \quad \{ \ \ldots \ \} \\
 & . \\
 & \quad \{ \ \ldots \ \} \\
 & . \\
 & \quad \{ \ \ldots \ \} \\
 & . \\
 & \quad \{ \ \ldots \ \} \\
 & ass_{n-1} \\
\equiv & \quad \{ \text{ ragione per cui } ass_{n-1} \equiv ass_n \ \} \\
 & ass_n
\end{array}$$

Il simbolo $\equiv$ viene utilizzato per denotare l'equivalenza tra asserzioni. Lo stesso formato di presentazione delle dimostrazioni verrà utilizzato per altre relazioni come l'implicazione tra asserzioni (denotata da $\Rightarrow$) o l'uguaglianza tra espressioni (denotata da $=$).

La giustificazione di ciascun passaggio è indicata tra parentesi graffe. Alcune comuni giustificazioni e la relativa notazione sono riferimenti a *definizioni* e *proprietà* { Definizione di ...} o { prop. ...}, *assunzioni* e *ipotesi locali* { **Ip:** ...}, semplici *calcoli aritmetici* { Aritmetica }, *regole o leggi logiche*, ad es. { Leibniz } o { Zero }.

La sintassi dei linguaggi sarà definita utilizzando grammatiche libere dal contesto, nella cosiddetta *Backus-Naur Form (BNF)*. Fissato un insieme di simboli non-terminali (*categorie sintattiche*) $NT_1, \ldots, NT_k$ e un alfabeto $\mathcal{A}$ di *simboli terminali*, le sentenze valide del linguaggio associato a ciascun simbolo non-terminale (sottoinsiemi di $\mathcal{A}^*$) sono definite da un insieme di produzioni del tipo:

$NT_i ::= \ < opt_1^i > \ | \ < opt_2^i > \ | \ \ldots \ | \ < opt_n^i >,$

[1] Rimandiamo comunque alle dispense di logica per una trattazione completa.

dove $< opt^i_j >$ sono stringhe di simboli terminali e non, e $i = 1, \ldots, k$. Intuitivamente la produzione associata al non-terminale NT_i è una regola (ricorsiva) che specifica in quali modi (il simbolo $|$ separa le differenti possibili alternative) una frase del linguaggio di NT_i può essere costruita a partire da componenti "più semplici".

Per esempio, i linguaggi delle cifre, dei numeri naturali e delle espressioni aritmentiche coinvolgenti numeri naturali, somma e prodotto sono definiti come segue:

```
Digit ::= 0 | 1 | ... | 9
Nat ::= Digit | Nat Digit
Exp ::= Nat | Exp × Exp | Exp + Exp
```

3 Il linguaggio

In questo paragrafo si introduce il linguaggio di programmazione studiato nelle note. Il linguaggio è detto *imperativo* in quanto fornisce un insieme di comandi espliciti che specificano azioni che devono essere eseguite dall'agente di calcolo e ne cambiano lo stato. Un problema di programmazione, in tali linguaggi, si pone proprio in termini della descrizione di uno stato *iniziale* (che rappresenta i *dati* del problema) e di uno stato *finale* (che rappresenta i *risultati* del problema). Lo soluzione di un problema siffatto consiste nella individuazione di una sequenza di azioni che modificano lo stato iniziale fino a trasformarlo nello stato finale desiderato.

Prima di formalizzare queste nozioni, per avere un'intuizione, possiamo pensare che lo stato dell'agente di calcolo durante l'esecuzione del programma sia il contenuto della memoria. Il comando fondamentale che permette di cambiare lo stato è il comando di assegnamento il cui effetto è quello di modificare il dato contenuto in una certa posizione della memoria. I comandi sono eseguiti in sequenza, uno dopo l'altro. Il flusso del controllo può, però, essere gestito mediante i costrutti di controllo più complessi che consentono, per esempio, di combinare due comandi in modo che l'uno o l'altro sia eseguito a seconda del verificarsi o meno di una certa condizione (costrutto condizionale) oppure permettono di ripetere un certo comando finché non si verifica una certa condizione (costrutto iterativo).

Noi considereremo un frammento di linguaggio di programmazione con comando vuoto, comando di assegnamento, sequenza, comando condizionale e comando iterativo. Per la sua semplicità esso non si presenta come un linguaggio reale, ma come uno strumento per lo studio dei concetti essenziali nella teoria dei linguaggi imperativi.

3.1 Espressioni

Il primo aspetto da affrontare nello studio di un linguaggio riguarda i meccanismi che esso mette a disposizione per rappresentare e manipolare i dati. Nel nostro caso i tipi di dato presenti nel linguaggio sono solamente valori booleani e interi ed i costrutti per manipolarli consentono di definire espressioni sia a valori booleani, che a valori interi. I valori di verità sono rappresentati sintatticamente dalle costanti booleane *true* e *false*, mentre i valori interi sono denotati tramite la loro rappresentazione decimale con segno.

In un linguaggio imperativo il calcolo procede attraverso l'elaborazione dello stato. È dunque fondamentale che le espressioni permettano di esprimere valori dipendenti dallo stato, o in altri termini che lo stato possa essere riferito nelle espressioni. Lo *stato* è astrattamente rappresentato mediante un insieme di associazioni tra nomi, detti *identificatori*, e valori. Formalmente, detta `Ide` la categoria sintattica degli identificatori, uno stato è una funzione:

$$\sigma : \texttt{Ide} \to \mathbb{Z} \cup \mathbb{B},$$

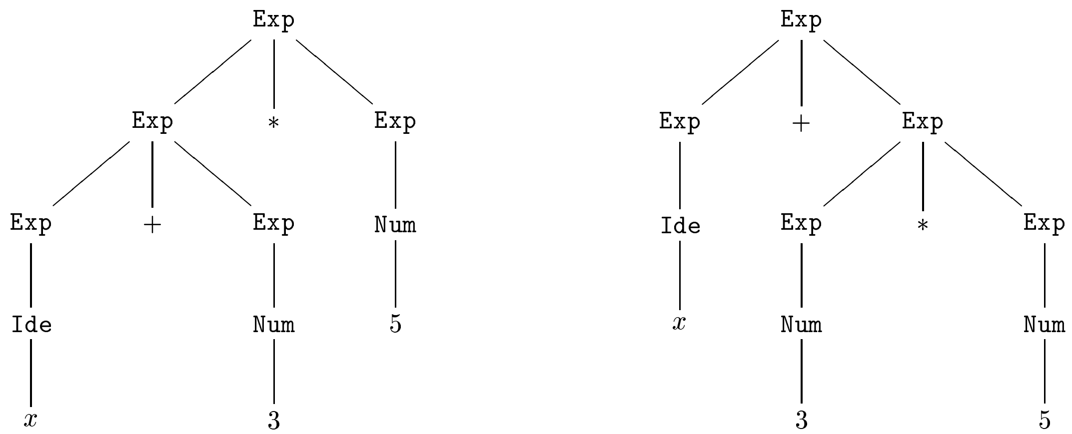
che associa a ciascun identificatore il proprio valore, ovvero un numero intero o un valore di verità. Indicheremo con *State* l'insieme di tutti gli stati.

Il linguaggio delle espressioni è definito dalla seguente grammatica BNF:

$\texttt{Exp} ::= \texttt{Const} \mid \texttt{Ide} \mid \texttt{(Exp)} \mid \texttt{Exp Op Exp} \mid \textit{not}\ \texttt{Exp.}$
$\texttt{Op} ::= + \mid - \mid * \mid \textit{div} \mid \textit{mod} \mid = \mid \neq \mid < \mid \leq \mid > \mid \geq \mid \textit{or} \mid \textit{and.}$
$\texttt{Const} ::= \texttt{Num} \mid \texttt{Bool.}$
$\texttt{Bool} ::= \textit{true} \mid \textit{false.}$

dove `Num` è la categoria sintattica che definisce i numeri interi in rappresentazione decimale.

Si noti che la grammatica considerata, non solo non assegna nessuna precedenza agli operatori, ma è anche ambigua. Ad esempio, l'espressione $x + 3 * 5$ può essere derivata in modi differenti:



Questo ha solo lo scopo di semplificare la definizione del linguaggio. Il processo di disambiguazione e di attribuzione di precedenze tra gli operatori richiede l'introduzione di ulteriori simboli non terminali nella grammatica e, pur essendo necessario quando si voglia implementare realmente il linguaggio, non ha un contenuto concettuale. L'ambiguità può essere un problema quando si danno definizioni induttive guidate dalla sintassi. La presenza della produzione `(Exp)` permette di scrivere espressioni non ambigue, quindi potremmo supporre di lavorare solo su stringhe aventi un unico albero di derivazione. Alternativamente, possiamo pensare di operare, non sulle stringhe di simboli terminali, ma sui corrispondenti alberi di derivazione e ossia di fare riferimento alla cosiddetta *sintassi astratta* del linguaggio.

Nel seguito utilizzeremo le seguenti convenzioni di notazione per semplificare la definizione della semantica del linguaggio:

- $n, m, n', m', \ldots$ indicano elementi della categoria sintattica `Num`;
- $b, b', \ldots$ indicano elementi della categoria sintattica `Bool`;
- $x, y, z, \ldots$ indicano elementi della categoria sintattica `Ide`;
- $E, E', \ldots$ indicano elementi della categoria sintattica `Exp`.

La semantica di un'espressione viene data mediante una *funzione di interpretazione semantica*

$$\mathcal{E} : \texttt{Exp} \times \mathit{State} \to \mathbb{Z} \cup \mathbb{B},$$

ovvero una funzione che data un'espressione ed uno stato fornisce il valore assunto dall'espressione in quello stato. La funzione $\mathcal{E}$ è definita per casi sulla struttura sintattica di un'espressione. Nella definizione di $\mathcal{E}$ verrà indicato convenzionalmente con **op** il significato (la funzione) che interpreta il generico simbolo di operatore *op* della sintassi. Ad esempio, se *op* è l'operatore $+$, allora **op** denota la funzione di somma $+ : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$; se *op* è l'operatore logico *and*, allora **op** denota il connettivo $\wedge$, e così via.

$$
\begin{array}{rcll}
\mathcal{E}(true, \sigma) & = & \mathbf{tt} & \\
\mathcal{E}(false, \sigma) & = & \mathbf{ff} & \\
\mathcal{E}(n, \sigma) & = & \mathbf{n} & \text{se} \quad n \in \texttt{Num} \\
\mathcal{E}(x, \sigma) & = & \sigma(x) & \text{se} \quad x \in \texttt{Ide} \\
\mathcal{E}(E \ op\ E', \sigma) & = & \mathcal{E}(E, \sigma)\ \mathbf{op}\ \mathcal{E}(E', \sigma) & \text{se} \quad op \in \{+, -, div, mod, =, \neq, <, >, \leq, \geq\} \\
 & & & \qquad \mathcal{E}(E, \sigma) \in \mathbb{Z} \ \text{ e } \ \mathcal{E}(E', \sigma) \in \mathbb{Z} \\
\mathcal{E}(E \ op\ E', \sigma) & = & \mathcal{E}(E, \sigma)\ \mathbf{op}\ \mathcal{E}(E', \sigma) & \text{se} \quad op \in \{and, or, =, \neq\} \\
 & & & \qquad \mathcal{E}(E, \sigma) \in \mathbb{B} \ \text{ e } \ \mathcal{E}(E', \sigma) \in \mathbb{B} \\
\mathcal{E}(not\ E, \sigma) & = & \neg\mathcal{E}(E, \sigma) & \text{se} \quad \mathcal{E}(E, \sigma) \in \mathbb{B} \\
\mathcal{E}((E), \sigma) & = & \mathcal{E}(E, \sigma) &
\end{array}
$$

Si noti che la funzione $\mathcal{E}$ è *parziale* sul suo dominio, ovvero esistono coppie (E, σ) tali che $\mathcal{E}(E, \sigma)$ non è definita. Abbiamo ad esempio, detto σ un generico stato:

$$
\begin{array}{ll}
 & \mathcal{E}(x \ div\ 0, \sigma) \\
= & \qquad\qquad \{\text{definizione di } \mathcal{E}\} \\
 & \mathcal{E}(x, \sigma)\ \mathbf{div}\ \mathcal{E}(0, \sigma) \\
= & \qquad\qquad \{\text{definizione di } \mathcal{E}\} \\
 & \sigma(x)\ \mathbf{div}\ 0
\end{array}
$$

Essendo la funzione **div** indefinita quando il divisore è nullo, il precedente calcolo non può essere portato a termine. Laddove necessario indicare esplicitamente che per una certa espressione E e per un certo stato σ, il valore di $\mathcal{E}(E, \sigma)$ non è definito, scriveremo $\mathcal{E}(E, \sigma) = \perp$. Ciò può essere visto come una estensione del codominio della funzione $\mathcal{E}$ all'insieme $\mathbb{Z} \cup \mathbb{B} \cup \{\perp\}$. Tale estensione richiede una analoga estensione del dominio e del codominio di ogni funzione **op** che interpreti il simbolo sintattico *op*. Ometteremo di indicare esplicitamente tale estensione.

3.2 Comandi

Il ruolo delle espressioni è dunque denotare valori (dipendenti dallo stato). Il ruolo dei *comandi* è, invece, *modificare*, con la loro esecuzione, lo stato.

Formalmente la sintassi dei *comandi* è definita dalla seguente grammatica:

Com ::= **skip** | Ide_List := Exp_List | Com ; Com | **if** Exp **then** Com **else** Com **fi** |
  while Exp **do** Com **endw**
Ide_list ::= Ide | Ide, Ide_List
Exp_List ::= Exp | Exp, Exp_List

Il comand **skip** è anche detto *comando vuoto*, in quanto non altera lo stato nel quale viene eseguito e termina. Un *assegnamento (multiplo)* $x_1, \ldots, x_n := E_1, \ldots, E_k$ assegna a ciascuna variabile x_i nella parte sinistra del comando il valore, valutato nello *stato corrente*, della corrispondente espressione E_i nella parte destra, quindi termina. L'esecuzione di una *sequenza* di comandi $C; C'$ consiste nell'esecuzione del comando C, seguita (qualora C termini) dall'esecuzione del comando C'. L'esecuzione di un *comando condizionale* **if** B **then** C **else** C' **fi**, inizia con la valutazione dell'espressione booleana B, detta *guardia*. Se B è vera, si esegue C, altrimenti (se B è falsa) si esegue C'. Infine, l'esecuzione di un *comando iterativo* **while** B **do** C **endw**, inizia con la valutazione dell'espressione booleana B, detta *guardia*. Se B è falsa, allora l'esecuzione del ciclo termina subito, altrimenti, se B è vera, si esegue il comando C, detto *corpo* del ciclo. Quando C termina, il processo è ripetuto.

La sintassi fornita non mette in evidenza alcuni vincoli, quali il fatto che, in un assegnamento del tipo $x_1, \ldots, x_n := E_1, \ldots, E_k$, gli identificatori a sinistra del simbolo := devono essere *distinti* tra loro ed il numero di identificatori e di espressioni deve essere lo stesso ($n = k$). Inoltre, nei linguaggi di programmazione reali, gli identificatori devono essere *dichiarati* prima di essere utilizzati nei comandi e nelle espressioni. Al momento della dichiarazione, ad essi viene associato un *tipo*, che definisce l'insieme dei valori che possono assumere durante l'esecuzione e le operazioni che su di essi possono essere eseguite. Ad esempio, il linguaggio Pascal prevede dichiarazioni come:

var x, y : **integer**
var b : **boolean**

grazie alle quali gli identificatori x, y e b diventano identificatori utilizzabili all'interno dei comandi e delle espressioni. Inoltre, la prima dichiarazione vincola gli identificatori x, y ad essere associati, nello stato, a valori di tipo intero, mentre la seconda vincola l'identificatore b ad essere associato a valori booleani. Noi non tratteremo qui, esplicitamente, la dichiarazione degli identificatori e supporremo che le espressioni che compaiono in un programma siano sempre ben tipate.

Vogliamo ora dare una descrizione più precisa, anche se ancora informale, del significato dei comandi del nostro linguaggio. La descrizione è di tipo operazionale, ovvero consiste in un insieme di regole, che specificano in che modo l'esecuzione di un comando C a partire da uno stato iniziale σ, causa la transizione in un nuovo stato σ'.

- L'esecuzione del *comando vuoto* **skip**, a partire da uno stato σ, causa una transizione nello stesso stato σ.
- L'esecuzione del *comando di assegnamento*

  $$x_1, \ldots, x_n := E_1, \ldots, E_n,$$

  a partire da uno stato σ, causa la transizione nel nuovo stato

  $$\sigma[{}^{\mathcal{E}(E_1,\sigma)}/_{x_1}, \ldots, {}^{\mathcal{E}(E_n,\sigma)}/_{x_n}],$$

  ottenuto associando a ciascuna variabile nella parte sinistra dell'assegnamento il valore, in σ, dell'espressione corrispondente. Tale stato è definito soltanto se, per ogni $i = 1, \ldots, n$, è definito il valore $\mathcal{E}(E_i, \sigma)$.
- L'esecuzione della *sequenza di comandi* $C; C'$, a partire da uno stato σ, causa la transizione nello stato σ' che si ottiene eseguendo il comando C' a partire dallo stato σ'' ottenuto per effetto dell'esecuzione di C nello stato σ.
- L'esecuzione del comando condizionale **if** E **then** C **else** C' **fi** a partire da uno stato σ causa la transizione nello stato σ' che si ottiene dall'esecuzione del comando C nello stato σ, se $\mathcal{E}(E, \sigma) = \mathbf{tt}$, e dall'esecuzione di C' nello stato σ, se $\mathcal{E}(E, \sigma) = \mathbf{ff}$.
- L'esecuzione del comando iterativo **while** E **do** C **endw** causa una transizione nello stesso stato σ, se $\mathcal{E}(E, \sigma) = \mathbf{ff}$. Se invece $\mathcal{E}(E, \sigma) = \mathbf{tt}$, l'esecuzione causa la transizione nello stato σ' ottenuto dall'esecuzione del comando **while** E **do** C **endw** a partire dallo stato σ'' ottenuto per effetto dell'esecuzione di C nello stato σ.

Dalle descrizioni precedenti è chiaro che esistono comandi che, a partire da uno stato σ, non causano alcuna transizione di stato. Tali situazioni corrispondono, in pratica, a situazioni di errore o a situazioni in cui l'esecuzione di un comando non ha termine. Consideriamo ad esempio il comando $x := 3$ **div** 0. Poiché $\mathcal{E}(3 \textbf{ div } 0, \sigma)$ non è definito, il comando non è in grado di transire in alcuno stato.

Si consideri invece l'esecuzione del comando **while** $x > 10$ **do** $x := x + 1$ **endw**, a partire da uno stato σ_0 in cui $\sigma_0(x) = \mathbf{15}$. La descrizione informale data in precedenza richiede di:

- determinare lo stato ottenuto eseguendo l'assegnamento $x := x + 1$ a partire dallo stato σ_0, che è chiaramente lo stato $\sigma_1 = \sigma_0[{}^{\mathbf{16}}/_x]$;

- determinare lo stato ottenuto eseguendo il comando **while** $x > 10$ **do** $x := x + 1$ **endw** a partire dal nuovo stato σ_1.

Per determinare quest'ultimo dobbiamo, di nuovo:

- determinare lo stato ottenuto eseguendo l'assegnamento $x := x + 1$ a partire dallo stato σ_1, che è chiaramente lo stato $\sigma_2 = \sigma_1[^{\mathbf{17}}/_x] = \sigma_0[^{\mathbf{16}}/_x][^{\mathbf{17}}/_x] = \sigma_0[^{\mathbf{17}}/_x]$;

- determinare lo stato ottenuto eseguendo il comando **while** $x > 10$ **do** $x := x + 1$ **endw** a partire dal nuovo stato σ_2.

È facile convincersi che tale procedimento non ha termine: il tentativo di determinare lo stato finale in cui transisce il comando, ci porta a determinare una sequenza infinita di stati.

Un'altra interpretazione informale del comando iterativo può essere data nel seguente modo. L'esecuzione del comando **while** E **do** C **endw** (dove supponiamo per semplicità che C non contenga a sua volta comandi iterativi) corrisponde all'esecuzione della sequenza (eventualmente vuota o infinita)

$$C_1; C_2; \ldots; C_n; \ldots,$$

dove ogni C_i coincide con C. Indichiamo con σ_0 lo stato iniziale e con σ_i lo stato risultante dall'esecuzione del comando C_i a partire dallo stato σ_{i-1}, per $i > 1$. Si hanno le due seguenti possibilità:

(a) la sequenza è finita, e cioè del tipo $C_1; C_2; \ldots; C_n$, con $n \geq 0$, e lo stato risultante dall'esecuzione del comando a partire da σ_0 è lo stato σ_n, tale che $n = \min\{i \geq 0 \mid \mathcal{E}(E, \sigma_i) = \mathbf{ff}\}$ (si noti che $n = 0$, ovvero il comando C non è eseguito neppure una volta, quando $\mathcal{E}(E, \sigma_0) = \mathbf{ff}$);

(b) la sequenza è infinita e, per ogni $i \geq 0$, si ha che $\mathcal{E}(E, \sigma_i) = \mathbf{tt}$.

La precedente descrizione informale diventa più complessa nel caso in cui C possa contenere a sua volta comandi iterativi.

Nel prossimo paragrafo daremo una descrizione formale della semantica del linguaggio dei comandi in stile *assiomatico*.

4 Semantica assiomatica

Nel paragrafo precedente abbiamo definito in modo intuitivo il significato dei costrutti del nostro linguaggio. Ora, la spiegazione fornita potrebbe sembrare sufficientemente chiara ed in effetti, per lungo tempo, ai linguaggi di programmazione è stata associata solo una semantica informale. Tuttavia questo stile si è rivelato, poi, fonte di errori e problemi, sia per l'implementazione dei linguaggi, sia per lo sviluppo ed il ragionamento sui singoli programmi scritti in un dato linguaggio di programmazione. Per eliminare questo pericolo, la spiegazione informale deve essere accompagnata (non sostituita!) da una definizione rigorosa della semantica. In queste note si esplora l'*approccio assiomatico* alla semantica, nel quale il significato dei costrutti di un linguaggio di programmazione viene specificato fornendo un sistema di regole che consentono di derivare proprietà dei programmi, espresse come *triple di Hoare*

$$\{Q\}\ C\ \{R\},$$

dove Q e R sono asserzioni e C è un comando del linguaggio. Il significato della tripla è: "a partire da ogni stato che soddisfi Q, l'esecuzione del comando C termina in uno stato che soddisfa R". Data una tripla $\{Q\}\ C\ \{R\}$, possiamo pensare che la coppia di asserzioni Q, R specifichi il comportamento atteso del programma. Dunque determinare C in modo che la tripla sia soddisfatta significa fornire un programma che soddisfa la specifica formale. Se invece anche il comando C è dato, la verifica della correttezza della tripla può essere vista come una verifica di correttezza del comando rispetto alla specifica. Queste considerazioni saranno approfondite ed esemplificate nel seguito.

Prima di introdurre la semantica assiomatica del nostro linguaggio occorre, però, definire con precisione cos'è un'asserzione e cosa significa che uno stato soddisfa una data asserzione.

4.1 Stati e asserzioni

Nel seguito indicheremo con $P, Q, R, \ldots$ formule ben formate del calcolo dei predicati, che chiameremo *asserzioni*. Le asserzioni vengono utilizzate per descrivere insiemi di stati. Ad esempio l'asserzione $x > 0 \wedge y = z$ descrive il sottinsieme proprio di *State* costituito da tutti e soli gli stati in cui il valore di x è strettamente positivo e in cui y e z hanno lo stesso valore. Il linguaggio in cui descriviamo le asserzioni può essere visto come un'estensione del linguaggio delle espressioni booleane (ovvero delle espressioni $E \in Exp$ tali che $\mathcal{E}(E, \sigma) \in \mathbb{B}$). La definizione induttiva è data come segue:

- un'espressione booleana B è un'asserzione;
- se P, Q sono asserzioni, anche $\neg P$, $P \wedge Q$, $P \vee Q$, $P \Rightarrow Q$, $P \equiv Q$ sono asserzioni
- se x è una variabile e P un'asserzione, anche $(\forall x.P)$ e $(\exists x.P)$ sono asserzioni.

Per snellire la notazione, quando un'espressione booleana del linguaggio verrà utilizzata all'interno di una asserzione, rimpiazzeremo implicitamente in essa le eventuali occorrenze degli operatori *and*, *or* e *not* con i corrispondenti connettivi $\wedge$, $\vee$ e $\neg$.

In un'asserzione del tipo $(\forall x.P)$ (risp. $(\exists x.P)$), la (sotto)asserzione P è detta la *portata* del quantificatore universale (risp. esistenziale). Data un'asserzione P ed un'occorrenza di variabile x in P, tale occorrenza è detta *legata* se occorre nella portata di un quantificatore del tipo $\forall x$ o $\exists x$, ed è detta *libera* altrimenti. Inoltre un'asserzione P è detta *chiusa* se P non contiene occorrenze libere di variabili, *aperta* altrimenti.

Nel seguito, data un'asserzione P, indicheremo con $\mathit{free}(P)$ l'insieme delle variabili che occorrono libere in P. Ad esempio $\mathit{free}(x > y \wedge z \leq 2) = \{x, y, z\}$. Dato una stato σ e un'asserzione P con $\mathit{free}(P) = \{x_1, \ldots, x_n\}$, utilizzeremo l'abbreviazione P^σ ad indicare la formula $P_{x_1,\ldots,x_n}^{\sigma(x_1),\ldots,\sigma(x_n)}$. In altre parole, P^σ sta per la formula ottenuta sostituendo in P ogni occorrenza libera di variabile con il corrispondente valore in σ.[2] Si noti che P^σ è un'asserzione chiusa.

[2]Assumiamo sempre che tutte le variabili libere in P abbiano un valore in σ. Inoltre, in queste note, P_x^t denota la formula ottenuta da P rmpiazzando tutte le occorrenze *libere* della variabile x con t

Esempio 4.1 Sia P la formula $(x \geq y \ \wedge \ z \leq x)$ e sia σ lo stato $\{x \mapsto 10, y \mapsto 5, z \mapsto 18, w \mapsto 3\}$. Allora P^σ è la formula $(10 \geq 5 \ \wedge \ 18 \leq 10)$.

Nel seguito indicheremo con $\mathit{free}(E)$ anche le variabili che compaiono in un'espressione E. Ad esempio $\mathit{free}(x + y - 3) = \{x, y\}$.

Si noti che, nell'esempio precedente, lo stato σ è rappresentato elencando tutte e sole le associazioni corrispondenti alle variabili il cui valore è definito in σ. In altri termini, il suddetto stato σ è la funzione:

$$\sigma(v) = \begin{cases} 10 & \text{se } v = x; \\ 5 & \text{se } v = y; \\ 18 & \text{se } v = z; \\ 3 & \text{se } v = w; \\ \text{indef.}, & \text{altrimenti.} \end{cases}$$

Nel seguito adotterremo spesso questa rappresentazione compatta per gli stati.

Definizione 4.2 Siano P un'asserzione e sia σ uno stato. Allora diciamo che σ *soddisfa* P, indicato con $\sigma \ \models \ P$, se vale P^σ (ovvero $P^\sigma \equiv \mathbf{tt}$). Analogamente, diciamo che σ *non soddisfa* P, indicato con $\sigma \ \not\models \ P$, se non vale P^σ (ovvero $P^\sigma \equiv \mathbf{ff}$). Formalmente:

$$\sigma \ \models \ P \ \equiv \ P^\sigma \qquad \text{e} \qquad \sigma \ \not\models \ P \ \equiv \ \neg P^\sigma.$$

□

Esempio 4.3 Siano P e σ come nell'esempio 4.1. Allora $\sigma \ \not\models \ P$. Infatti:

$$\begin{array}{ll} & \sigma \ \models \ P \\ \equiv & \qquad \{ \text{ definizione 4.2 } \} \\ & (10 \geq 5 \ \wedge \ 18 \leq 10) \\ \equiv & \qquad \{ \ \neg(18 \leq 10), \text{Zero } \} \\ & \mathbf{ff} \end{array}$$

Detto σ' lo stato $\{x \mapsto 10, y \mapsto 5, z \mapsto 1\}$, abbiamo $\sigma' \ \models \ P$. Infatti:

$$\begin{array}{ll} & \sigma' \ \models \ P \\ \equiv & \qquad \{ \text{ definizione 4.2 } \} \\ & (10 \geq 5 \ \wedge \ 1 \leq 10) \\ \equiv & \qquad \{ \ 10 \geq 5, 1 \leq 10, \text{Idempotenza } \} \\ & \mathbf{tt} \end{array}$$

Data una asserzione P, denotiamo con $\{P\}$ il sottinsieme di State definito come segue:

$$\{P\} = \{\sigma \in \mathit{State} \ \mid \ \sigma \ \models \ P\}.$$

Nel seguito utilizzeremo le seguenti proprietà che riguardano la relazione tra stati e asserzioni.

Proprietà 4.4 Siano σ, σ' stati, E un'espressione e P un'asserzione.

(i) se per ogni $x \in \mathit{free}(E)$, $\sigma(x) = \sigma'(x)$, allora $\mathcal{E}(E, \sigma) = \mathcal{E}(E, \sigma')$

(ii) se per ogni $x \in \mathit{free}(P)$, $\sigma(x) = \sigma'(x)$, allora $\sigma \ \models \ P$ se e solo se $\sigma' \ \models \ P$

(iii) per ogni variabile x, vale $\sigma[^{\mathcal{E}(E,\sigma)}/_x] \ \models \ P$ se e solo se $\sigma \ \models \ P^E_x$.

□

Omettiamo la dimostrazione delle proprietà precedenti e diamo due esempi della (iii).

Esempio 4.5

$$
\begin{array}{ll}
(1) & \sigma[^{\mathcal{E}(z+1,\sigma)}/_x] \models (x > y) \\
\equiv & \quad \{ \text{ definizione 4.2 e definizione di } \mathcal{E} \ \} \\
 & (\sigma(z) + 1) > \sigma(y) \\
\equiv & \quad \{ \text{ definizione di } \models \ \} \\
 & \sigma \models (z + 1) > y \\
\equiv & \quad \{ \text{ sostituzione } \} \\
 & \sigma \models (x > y)_x^{z+1}
\end{array}
$$

$$
\begin{array}{ll}
(2) & \sigma[^{\mathcal{E}(x+y,\sigma)}/_x] \models (x = z) \\
\equiv & \quad \{ \text{ definizione 4.2 e definizione di } \mathcal{E} \ \} \\
 & (\sigma(x) + \sigma(y)) = \sigma(z)) \\
\equiv & \quad \{ \text{ definizione di } \models \ \} \\
 & \sigma \models (x + y) = z \\
\equiv & \quad \{ \text{ sostituzione } \} \\
 & \sigma \models (x = z)_x^{x+y}
\end{array}
$$

4.2 Triple di Hoare

La semantica *assiomatica* dei linguaggi di programmazione si basa sul concetto di formula di correttezza o *tripla di Hoare*. Una tripla di Hoare è un asserto rappresentato dalla terna

$$\{Q\}\ C\ \{R\}$$

dove C è un comando e Q, R sono asserzioni, dette rispettivamente *precondizione* e *postcondizione*. Il significato intuitivo di una tripla come la precedente si può riassumere dicendo che essa è soddisfatta se valgono i seguenti due fatti: per ogni stato σ che soddisfa Q

(1) l'esecuzione del comando C, a partire dallo stato σ, termina;

(2) detto σ' lo stato in cui termina l'esecuzione di C a partire da σ, σ' soddisfa R.

Una tripla $\{P\}\ C\ \{R\}$ può essere letta in modi diversi.

Semantica. Dati P e C, determinare l'asserzione R in modo che sia verificata la $\{P\}\ C\ \{R\}$ è un modo per descrivere il comportamento di C. Ad esempio, dati la precondizione $(x = 1)$ ed il comando $x := x + 1$, la postcondizione $(x = 2)$ descrive correttamente il comportamento di C.

Specifica. Dati P e R, determinare un comando C tale che $\{P\}\ C\ \{R\}$ significa risolvere il problema di programmazione specificato dalla pre e dalla postcondizione date. Ad esempio, con $P \equiv (x \neq 0)$ e $R \equiv (z = y\ div\ x)$, un comando che verifica la tripla è dato da $z := y\ div\ x$. (Si noti che la precondizione garantisce che il calcolo di $y\ div\ x$ dà luogo ad un valore.) Nel prosieguo vedremo come il sistema di dimostrazione di formule di correttezza suggerisca anche una metodologia per la derivazione di comandi che verificano una specifica.

Correttezza. Dati P, C e R, dimostrare che la tripla $\{P\}\ C\ \{R\}$ è soddisfatta corrisponde ad una verifica di correttezza del comando C rispetto alla specifica determinata da P e R.

La dimostrazione di formule di correttezza si basa su un insieme di assiomi e regole di inferenza (proof system) che daremo nel seguito. La notazione che utilizziamo per descrivere le regole è la seguente

$$\boxed{\dfrac{\phi_1 \ldots \phi_n}{\phi} \quad \text{dove " \ldots "}}$$

in cui ϕ è una *formula di correttezza*, $\phi_1, \ldots, \phi_n$ sono *asserzioni* o *formule di correttezza* e la clausola a margine indica le eventuali *condizioni di applicabilità* della regola.

Nel sistema di prova per le formule di correttezza diamo per scontate tutte le leggi del calcolo proposizionale e dei predicati.

La prima regola riguarda una proprietà generale delle triple.

$$\frac{P \Rightarrow P' \qquad \{P'\}\ C\ \{R'\} \qquad R' \Rightarrow R}{\{P\}\ C\ \{R\}} \quad (pre-post)$$

Il significato della regola è immediato ricordando che:

(i) $P \Rightarrow P'$ significa che l'insieme $\{P\}$ è un sottinsieme di $\{P'\}$

(ii) $R' \Rightarrow R$ significa che l'insieme $\{R'\}$ è un sottinsieme di $\{R\}$

(iii) $\{P'\}\ C\ \{R'\}$ significa che l'esecuzione di C, a partire da uno stato in $\{P'\}$ termina in uno stato in $\{R'\}$.

La conclusione della regola stabilisce che l'esecuzione di C, a partire da uno stato in $\{P\}$, e dunque in $\{P'\}$ per (i), termina grazie a (iii) in uno stato in $\{R'\}$, e dunque in $\{R\}$ per (ii).

La regola $(pre-post)$ stabilisce che, in una tripla di Hoare, si può rafforzare la precondizione ed indebolire la post-condizione. Casi particolari di tale regola sono le due regole seguenti:

$$\frac{P \Rightarrow P' \qquad \{P'\}\ C\ \{R\}}{\{P\}\ C\ \{R\}} \quad (pre)$$

$$\frac{\{P\}\ C\ \{R'\} \qquad R' \Rightarrow R}{\{P\}\ C\ \{R\}} \quad (post)$$

La regola (pre) si ottiene dalla regola $(pre-post)$ con $R' = R$; la regola $(post)$ si ottiene da $(pre-post)$ con $P' = P$. In entrambi i casi si sfrutta la nota legge $p \Rightarrow p$.

Veniamo agli assiomi e alle regole per i comandi del linguaggio.

4.3 Comando vuoto

Il comando vuoto non modifica lo stato e termina, dunque affinché un'asserzione P sia soddisfatta dopo l'esecuzione di **skip** , la stessa asserzione deve essere soddisfatta prima dell'esecuzione del comando stesso. Il significato del comando vuoto è quindi dato dal seguente assioma:

$$\{P\}\ \textbf{skip}\ \{P\} \qquad (skip)$$

4.4 Comando di assegnamento

Limitiamoci dapprima al comando di assegnamento semplice (del tipo $x := E$). Sappiamo che l'effetto di tale assegnamento, a partire da uno stato σ è di transire nello stato $\sigma[^{\mathcal{E}(E,\sigma)}/_x]$, in cui l'unica associazione modificata è quella dell'identificatore x, al quale viene assegnato il valore $\mathcal{E}(E,\sigma)$ assunto dall'espressione E nello stato di partenza. Consideriamo allora una postcondizione R, uno stato iniziale σ ed il comando $x := E$ e supponiamo che $\mathcal{E}(E,\sigma)$ sia definita. Se vogliamo garantire che R valga per x dopo l'assegnamento, basterà assicurarci che R valga per E prima dell'assegnamento. Abbiamo, infatti:

$$
\begin{array}{ll}
 & \sigma[^{\mathcal{E}(E,\sigma)}/_x] \models R \\
\equiv & \quad \{ \text{ proprietà 4.4(iii) } \} \\
 & \sigma \models R^E_x
\end{array}
$$

Possiamo dunque formulare un assioma per l'assegnamento semplice del tipo:

$$\{P^E_x\}\ x := E\ \{P\}$$

Tale assioma assicura, ad esempio, che l'esecuzione del comando $x := y+1$ porta in uno stato che soddisfa l'asserzione $x > z$, a patto che venga eseguito a partire da uno stato che soddisfa l'asserzione $y + 1 > z$. Osserviamo tuttavia, che questa formulazione non tiene conto del fatto che una tripla di Hoare deve anche garantire che il comando coinvolto termini. Consideriamo ad esempio un assegnamento del tipo $x := y\ div\ z$ e la postcondizione $\{\mathbf{tt}\}$. L'assioma dato sopra viene istanziato in questo caso nella tripla

$\{\mathbf{tt}\}\ x := y\ div\ z\ \{\mathbf{tt}\}$

che chiaramente non è corretta, dal momento che solo gli stati σ tali che $\sigma \models z \neq 0$ garantiscono la terminazione del comando. Dunque la precondizione della tripla deve essere rafforzata imponendo delle ulteriori condizioni sullo stato iniziale, che garantiscano che l'espressione coinvolta nell'assegnamento abbia un valore nello stato iniziale (nell'esempio $z \neq 0$).

Definiamo a questo proposito una funzione *def* che, data un'espressione $E \in Exp$, calcola un'asserzione $def(E)$ che soddisfa la seguente proprietà: per ogni stato σ, $\sigma \models def(E)$ se e solo se, esiste v tale che $\mathcal{E}(E,\sigma) = v$. La definizione di tale funzione è data per casi sulla struttura di un'espressione.

$$
\begin{array}{rcll}
def(c) & = & \mathbf{tt} & \text{se} \quad c \in \texttt{Num} \text{ o } c \in \texttt{Bool} \\
def(x) & = & \mathbf{tt} & \text{se} \quad x \in \texttt{Ide} \\
def(E\ op\ E') & = & def(E) \wedge def(E') & \text{se} \quad op \in \left\{ \begin{array}{c} +,-,<,>,\leq,\geq \\ =,\neq, and, or \end{array} \right\} \\
def(E\ op\ E') & = & def(E) \wedge def(E') \wedge E' \neq 0 & \text{se} \quad op \in \{div, mod\} \\
def(not\ E) & = & def(E) & \\
def((E)) & = & def(E) &
\end{array}
$$

Si noti che manca un controllo della correttezza dei tipi dei dati manipolati dalle operazioni. Si suppone, infatti, che le espressioni che compaiono nei nostri programmi siano sempre ben tipate. L'idea è che il controllo dei tipi può avvenire in una fase preliminare e può essere considerato parte del processo di individuazione dei programmi legali del linguaggio.

Esempio 4.6 Consideriamo l'espressione $x\ div\ (z - y)$. Abbiamo:

$$
\begin{array}{ll}
 & def(x\ div\ (z-y)) \\
\equiv & \quad \{ \text{ definizione di } def() \ \}
\end{array}
$$

$$
\begin{array}{ll}
 & def(x) \wedge def((z-y)) \wedge z-y \neq 0 \\
\equiv & \quad \{ \text{ definizione di } def(), z-y \neq 0 \equiv z \neq y \ \} \\
 & \mathbf{tt} \wedge def(z-y) \wedge z \neq y \\
\equiv & \quad \{ \text{ definizione di } def(),\ \textit{Unità} \ \} \\
 & def(z) \wedge def(y) \wedge z \neq y \\
\equiv & \quad \{ \text{ definizione di } def() \ \} \\
 & \mathbf{tt} \wedge \mathbf{tt} \wedge z \neq y \\
\equiv & \quad \{ \ \textit{Unità} \ \} \\
 & z \neq y
\end{array}
$$

Siamo ora in grado di definire correttamente l'assioma per l'assegnamento semplice che diventa:

$$
\boxed{\{def(E) \wedge P_x^E\}\ x := E\ \{P\} \qquad (ass)}
$$

Si osservi che, nell'assioma, si assume implicitamente che la sostituzione di E per x in P sia lecita, ovvero che non succeda che variabili libere in E siano "catturate" da quantificatori in P_x^E. Per evitare che questo accada l'operazione di sostituzione effettua le ridenominazioni necessarie. Per esempio la tripla

$$\{\forall x \in \mathbb{N}.\ x \leq x\}\ y := x\ \{\forall x \in \mathbb{N}.\ y \leq x\},$$

non è un'istanza corretta dell'assioma (ass), mentre lo è

$$\{\forall z \in \mathbb{N}.\ x \leq z\}\ y := x\ \{\forall z \in \mathbb{N}.\ y \leq z\}.$$

Nel caso generale di *assegnamento multiplo*, vi sono più identificatori, a ciascuno dei quali deve essere assegnato il valore della corrispondente espressione, calcolata nello stato nel quale si esegue il comando. È importante ricordare che gli assegnamenti sono fatti tutti insieme, *dopo* aver calcolato il valore delle espressioni (che quindi è indipendente dall'ordine di valutazione). L'assioma per l'assegnamento multiplo, indicato ancora con (ass), è dunque l'ovvia estensione dell'assioma precedente.

$$
\boxed{\{def(E_1) \wedge \ldots \wedge def(E_k) \wedge P_{x_1,\ldots,x_k}^{E_1,\ldots,E_k}\}\ x_1,\ldots,x_k := E_1,\ldots,E_k\ \{P\} \qquad (ass)}
$$

Esempio 4.7 Si verifichi la tripla

$\{x = n \wedge y = m\}\ x, y := y, x\ \{x = m \wedge y = n\}$

La verifica è immediata dal momento che, essendo $def(x) \equiv def(y) \equiv \mathbf{tt}$, la tripla è un'istanza di (ass).

Esempio 4.8 Si verifichi la tripla

$\{x = 5\}\ x := x + 1\ \{x > 2\}$

Applicando (ass), e osservando $def(x + 1) \equiv \mathbf{tt}$ abbiamo:

$\{x + 1 > 2\}\ x := x + 1\ \{x > 2\}$

Poiché assumendo $x = 5$ si ha:

$$
\begin{array}{ll}
 & x + 1 > 2 \\
\equiv & \quad \{ \text{ calcolo } \} \\
 & x > 1 \\
\equiv & \quad \{ \ \mathbf{Ip:}\ x = 5 \ \} \\
 & 5 > 1 \\
\equiv & \quad \{ \text{ def. di } > \} \\
 & \mathbf{tt}
\end{array}
$$

Possiamo concludere che $x = 5 \Rightarrow x + 1 > 2$, e quindi, grazie alla regola (*pre*), la correttezza della tripla data.

Esempio 4.9 Si verifichi la tripla:

$$\{s = (\sum i \in [0, x).\ i)\}\ x, s := x + 1, s + x\ \{s = (\sum i \in [0, x).\ i)\}$$

Applicando (*ass*) e osservando $def(x + 1) \equiv def(s + x) \equiv \mathbf{tt}$, otteniamo

$$\{s + x = (\sum i \in [0, x + 1).\ i)\}\ x, s := x + 1, s + x\ \{s = (\sum i \in [0, x).\ i)\}$$

Infine:

$$\begin{array}{ll} & s + x = (\sum i \in [0, x + 1).\ i) \\ \equiv & \quad \{\ (\sum i \in [0, x + 1).\ i) = (\sum i \in [0, x).\ i) + x\ \} \\ & s + x = (\sum i \in [0, x).\ i) + x \\ \equiv & \quad \{\ \text{calcolo}\ \} \\ & s = (\sum i \in [0, x).\ i) \end{array}$$

Si noti che quest'ultima asserzione coincide con la precondizione della tripla data, che quindi è direttamente un'istanza di (*ass*).

4.5 Sequenza di comandi

La regola per derivare la correttezza di una tripla del tipo $\{P\}\ C; C'\ \{Q\}$ esprime il fatto che il comando C, a partire da uno stato in $\{P\}$ deve portare in uno stato a partire dal quale l'esecuzione di C' porti in uno stato che soddisfa Q. Dunque:

$$\boxed{\frac{\{P\}\ C\ \{R\} \qquad \{R\}\ C'\ \{Q\}}{\{P\}\ C; C'\ \{Q\}}\ (seq)}$$

Esempio 4.10 Si verifichi la correttezza della seguente tripla:

$$\{x \geq y - 1\}\ x := x + 1; y := y - 1\ \{x > y\}.$$

Per poter applicare la regola (*seq*) per la sequenza di comandi dobbiamo provare che valgono le premesse della regola, ovvero

(1) $\{R\}\ y := y - 1\ \{x > y\}$

(2) $\{x \geq y - 1\}\ x := x + 1\ \{R\}$

per un'opportuna asserzione R. Ora, applicando la regola *(ass)*, dato che $def(y - 1) = \mathbf{tt}$, otteniamo che vale:

$$\{x > y - 1\}\ y := y - 1\ \{x > y\}. \qquad (\dagger)$$

Inoltre, applicando nuovamente la regola dell'assegnamento *(ass)* al primo comando si ottiene:

$$\{x + 1 > y - 1\}\ x := x + 1\ \{x > y - 1\}. \qquad (\dagger\dagger)$$

È immediato verificare che $x + 1 > y - 1$ è equivalente a $x \geq y - 1$ e quindi dalle premesse ($\dagger$) e ($\dagger\dagger$), applicando la regola *(ass)* si conclude $\{x \geq y - 1\}\ x := x + 1; y := y - 1\ \{x > y\}$.

Esempio 4.11 Determinare E affinché sia corretta la tripla

$$\{x = n \wedge y = m\}\ t := E; x := y; y := t\ \{x = m \wedge y = n\}$$

Trattandosi di una sequenza di comandi dobbiamo appellarci alla regola (*seq*) e determinare R che verifichi le triple

(1) $\{R\}\ x := y; y := t\ \{x = m \wedge y = n\}$

(2) $\{x = n \wedge y = m\}\ t := E\ \{R\}$

Essendo (1) a sua volta una sequenza dobbiamo determinare R' tale che

(1.1) $\{R'\}\ y := t\ \{x = m \wedge y = n\}$

(1.2) $\{R\}\ x := y\ \{R'\}$

Per (1.1), applicando (ass) e osservando $def(t) \equiv \mathbf{tt}$, abbiamo

$\{x = m \wedge t = n\}\ y := t\ \{x = m \wedge y = n\}$, ovvero $R' \equiv (x = m \wedge t = n)$.

Dunque (1.2) diviene

$\{R\}\ x := y\ \{x = m \wedge t = n\}$

e, applicando di nuovo (ass) abbiamo

$\{y = m \wedge t = n\}\ x := y\ \{x = m \wedge t = n\}$.

Concludiamo allora che, in (1), una soluzione per R è $y = m \wedge t = n$. Infine, dobbiamo determinare E in modo che sia soddisfatta (b), ovvero:

$\{x = n \wedge y = m\}\ t := E\ \{y = m \wedge t = n\}$.

È evidente che, con x per E, la precedente tripla diviene un'istanza di (ass).

4.6 Comando condizionale

Per un comando condizionale del tipo **if** E **then** C **else** C' **fi** la regola tiene conto delle due possibili evoluzioni dell'esecuzione di tale comando, che corrispondono al caso in cui, nello stato iniziale, l'espressione E abbia valore **tt** o **ff**, rispettivamente. Come nel caso del comando di assegnamento, si deve imporre che, nello stato iniziale, l'espressione E sia definita e quindi dia luogo ad un valore.

$$\frac{P \Rightarrow def(E) \qquad \{P \wedge E\}\ C\ \{Q\} \qquad \{P \wedge \neg E\}\ C'\ \{Q\}}{\{P\}\ \textbf{if}\ E\ \textbf{then}\ C\ \textbf{else}\ C'\ \textbf{fi}\ \{Q\}}\ (if)$$

Esempio 4.12 Si verifichi la tripla:

$\{x \geq 0\ \wedge\ y > 0\}$
  if $x\ div\ y > 0$
    then $z := x$
    else $z := y$
  fi
$\{z = \max(x, y)\}$

Osserviamo in primo luogo che la precondizione assicura la definizione della guardia, ovvero vale:
$x \geq 0\ \wedge\ y > 0 \quad \Rightarrow \quad def(x\ div\ y > 0)$

Restano da provare le triple corrispondenti ai rami **then** e **else** . Per quanto riguarda la prima, osserviamo che

$x \geq 0\ \wedge\ y > 0\ \wedge\ x\ div\ y > 0$
$\Rightarrow$ $\{\ x = (x\ div\ y) * y + r$ con $0 \leq r < y$ e calcolo $\}$
$x \geq y$
$\Rightarrow$ $\{$ Defin. di max $\}$
$x = \max(x, y)$
$\equiv$ $\{$ Sostituzione $\}$
$(z = \max(x, y))^x_z$

Inoltre per la regola dell'assegnamento (ass), vale la tripla:

$$\{(z = \max(x, y))^x_z\}\ z := x\ \{z = \max(x, y)\}$$

e quindi, per la regola (pre) si conclude che vale:

$$\{x \geq 0\ \wedge\ y > 0\ \wedge\ x\ div\ y > 0\}\ z := x\ \{z = \max(x, y)\}$$

Allo stesso modo, per il ramo **else** , si prova che vale

$$\{x \geq 0\ \wedge\ y > 0\ \wedge\ \neg(x\ div\ y > 0)\}\ z := y\ \{z = \max(x, y)\}$$

e quindi si conclude applicando la regola (if).

4.7 Comando iterativo

Il comando iterativo **while** E **do** C **endw** permette di esprimere l'iterazione, ossia l'esecuzione ripetuta, di un comando C fino al verificarsi di una condizione di terminazione E. La definizione della regola per tale comando richiede una certa cautela. Nella semantica informale abbiamo visto che l'esecuzione di **while** E **do** C **endw** corrisponde all'esecuzione di una sequenza del tipo

$$C_1; C_2; \ldots; C_n; \ldots$$

dove ogni C_i coincide con C ed n è il (minimo) numero di iterazioni necessario per falsificare la guardia E. Il problema è che, in generale, non è possibile stabilire a priori né se il comando termina (la sequenza è finita) né, qualora il comando termini, quale sia la lunghezza della sequenza (ovvero quante volte venga ripetuto il comando C). Se fossimo in grado di stabilire che il comando corrisponde ad una sequenza finita del tipo

$$\underbrace{C; C; \ldots; C}_{k\ volte}$$

potremmo appellarci a ripetute applicazioni della regola (seq) e stabilire una regola del tipo

$$\frac{\{P\}\ C\ \{P_1\} \qquad \{P_1\}\ C\ \{P_2\} \quad \ldots \quad \{P_{k-1}\}\ C\ \{Q\}}{\{P\}\ \textbf{while}\ E\ \textbf{do}\ C\ \textbf{endw}\ \{Q\}}$$

L'impossibilità di conoscere il numero k di ripetizioni del corpo di un comando iterativo del tipo **while** E **do** C **endw**, porta all'idea di determinare un'asserzione che sia vera indipendentemente da k e che, al tempo stesso sia significativa: sia Inv (per *invariante*) una asserzione con questa proprietà. Supponendo per il momento che il comando **while** E **do** C **endw** termini, possiamo allora concludere che vale una tripla del tipo

$$\{Inv\}\ \textbf{while}\ E\ \textbf{do}\ C\ \textbf{endw}\ \{Inv\}. \tag{1}$$

In realtà, possiamo dire qualcosa in più sulla post–condizione della tripla precedente, in quanto la semantica del comando garantisce che, nello stato finale, la guardia non è verificata. Detto altrimenti, possiamo riscrivere la (1) come:

$$\{Inv\}\ \textbf{while}\ E\ \textbf{do}\ C\ \textbf{endw}\ \{Inv \wedge \neg E\}. \tag{2}$$

Un modo per garantire che l'asserzione Inv sia effettivamente un'invariante per il comando **while** E **do** C **endw**, è richiedere che essa rimanga soddisfatta dopo una generica iterazione del comando stesso. Ciò significa che, a partire da un qualunque stato in cui vale Inv ed in cui la guardia E è soddisfatta (stato che corrisponde ad una generica iterazione del corpo), l'esecuzione di C deve portare in uno stato in cui l'invariante è ancora soddisfatta. Formalmente:

$$\{Inv \wedge E\}\ C\ \{Inv\}. \tag{3}$$

Se la tripla (3) è soddisfatta, la proprietà Inv vale dopo ogni iterazione del corpo. Dunque, supponendo che il comando **while** E **do** C **endw** termini, possiamo formulare una regola del tipo:

$$\frac{\{Inv \wedge\ E\}\ C\ \{Inv\}}{\{Inv\}\ \textbf{while}\ E\ \textbf{do}\ C\ \textbf{endw}\ \{Inv \wedge \neg E\}}$$

Dobbiamo ora trovare un modo per garantire la terminazione dell'intero comando. Due sono le fonti di possibile non terminazione del comando **while** E **do** C **endw**:

(i) la valutazione dell'espressione E ad ogni iterazione;

(ii) il fatto che E sia vera ad ogni iterazione.

Per quanto riguarda (i) basta assicurare che, prima di ogni iterazione, valga la condizione $def(E)$. A patto di garantire questo fatto anche nello stato iniziale, ciò si ottiene modificando la tripla (3) come segue:

$$\{Inv \wedge E\}\ C\ \{Inv \wedge def(E)\}. \tag{4}$$

Per quanto riguarda (ii), consideriamo la sequenza (eventualmente infinita) di stati $S_{E,C} = \sigma_0, \sigma_1, \ldots, \sigma_k, \ldots$ che corrispondono alle varie iterazioni del corpo C nell'esecuzione di **while** E **do** C **endw**. Più precisamente:

- σ_0 è lo stato iniziale;
- per $i > 0$, σ_i è lo stato che risulta dall'esecuzione del comando C nello stato σ_{i-1}.

Osserviamo che si possono presentare i due casi seguenti:

- il comando **while** E **do** C **endw** non termina, ovvero $S_{E,C}$ è una sequenza infinita e, per ogni σ_i in $S_{E,C}$, si ha $\sigma_i \models E$;
- il comando **while** E **do** C **endw** termina, ovvero $S_{E,C}$ è del tipo $\sigma_0, \sigma_1, \ldots, \sigma_{k-1}, \sigma_k$, per ogni $i < k$ si ha $\sigma_i \models E$ e infine $\sigma_k \models \neg E$.

Dunque la terminazione del comando **while** E **do** C **endw** può essere dimostrata garantendo che la sequenza $S_{E,C}$ è finita. A questo proposito, supponiamo di associare ad ogni stato σ_i in $S_{E,C}$ un numero naturale a_i, ottenendo così una successione di numeri naturali $\mathcal{A} = a_0, a_1, \ldots, a_k, \ldots$. È evidente che $S_{E,C}$ è finita se e soltanto se lo è la successione $\mathcal{A}$. Un modo per garantire quest'ultima proprietà è dimostrare che $\mathcal{A}$ è strettamente decrescente (ogni successione strettamente decrescente di numeri naturali è finita). Ogni elemento a_i di tale successione deve essere funzione dello stato σ_i corrispondente. Sia allora f una funzione, detta *funzione di terminazione*, che permette di ottenere il valore a_i corrispondente al generico stato σ_i in $S_{E,C}$. Poiché ogni stato σ_i nella sequenza soddisfa *Inv*, possiamo considerare una funzione f tale che:

$$(\sigma \models Inv) \Rightarrow f(\sigma) \geq 0 \tag{5}$$

La decrescenza della successione $\mathcal{A} = \{f(\sigma_0), f(\sigma_1), \ldots\}$ può essere ottenuta dimostrando che vale la seguente proprietà:

$$f(\sigma_{i-1}) > f(\sigma_i), \text{ per } i \geq 1. \tag{6}$$

Se la funzione f è del tipo $f(\sigma) = t$, dove t è un'espressione che, in generale, coinvolge i valori delle variabili di E e C, possiamo esprimere la (6) e la (5) come segue:

$$Inv \Rightarrow t \geq 0 \tag{7}$$

$$\{Inv \wedge E \wedge t = V\}\ C\ \{t < V\} \tag{8}$$

Si noti che, nella (8), la precondizione garantisce la descrescenza solo tra uno stato nella sequenza che soddisfa E e lo stato successivo (in altre parole, ci disinteressiamo dello stato finale della sequenza).

Si osservi ancora che V non è una variabile di programma. Rappresenta un valore generico assunto dalla funzione di terminazione ed è indispensabile in (8) per poter riferire il valore assunto da t prima dell'esecuzione di C, nella post-condizione $t < V$. Variabili come V sono dette *variabili di specifica* ed il loro ruolo sarà trattato estensivamente nel Paragrafo 6.

Quanto detto sin qui può essere allora riassunto nella seguente regola per il comando iterativo:

$$\frac{\begin{array}{c} Inv \Rightarrow t \geq 0 \\ \{Inv \wedge E\}\ C\ \{Inv \wedge def(E)\} \qquad \{Inv \wedge E \wedge t = V\}\ C\ \{t < V\} \end{array}}{\{Inv \wedge def(E)\}\ \textbf{while}\ E\ \textbf{do}\ C\ \textbf{endw}\ \{Inv \wedge \neg E\}} \quad (while)$$

La premessa $Inv \Rightarrow t \geq 0$ è detta *ipotesi di terminazione*, la premessa $\{Inv \wedge E\}\ C\ \{Inv \wedge def(E)\}$ *ipotesi di invarianza* e infine $\{Inv \wedge E \wedge t = V\}\ C\ \{t < V\}$ è detta *ipotesi di progresso.*

Una versione equivalente della regola precedente, che si ottiene applicando la regola $(pre - post)$, è la seguente:

$$\frac{\begin{array}{c} P \Rightarrow Inv \wedge def(E) \qquad Inv \wedge \neg E \Rightarrow Q \qquad Inv \Rightarrow t \geq 0 \\ \{Inv \wedge E\}\ C\ \{Inv \wedge def(E)\} \qquad \{Inv \wedge E \wedge t = V\}\ C\ \{t < V\} \end{array}}{\{P\}\ \textbf{while}\ E\ \textbf{do}\ C\ \textbf{endw}\ \{Q\}} \quad (while)$$

In pratica, succede spesso che le premesse della regola *(while)* siano soddisfatte, a parte l'implicazione $P \Rightarrow Inv \wedge def(E)$. Si noti che tale premessa richiede essenzialmente che lo stato iniziale in cui viene intrapreso il ciclo soddisfi l'asserzione invariante (oltre che la definizione della guardia). La regola (seq) suggerisce, in questi casi, di premettere al comando iterativo un comando cosiddetto di *inizializzazione* C_I che verifichi la tripla

$\{P\}\ C_I\ \{Inv \wedge def(E)\}$

come vedremo negli esempi che seguono.

Esempio 4.13 Sia C il comando

while $x < n$ **do**
$\quad x, s := x + 1, s + x$
endw

e sia Inv l'asserzione:

$Inv : s = (\sum i \in [0, x).\ i) \wedge 0 \leq x \leq n.$

Verifichiamo che Inv è effettivamente invariante per il comando dato, ovvero che è verificata la tripla:

$\{Inv \wedge x < n\}\ x, s := x + 1, s + x\ \{Inv \wedge def(x < n).\}$ (†)

Osserviamo innanzitutto che $def(x < n)$ si può omettere, essendo $def(x < n) \equiv \mathbf{tt}$. La dimostrazione dell'esempio 4.9 stabilisce che la tripla

$\{s = (\sum i \in [0, x).\ i)\}\ x, s := x + 1, s + x\ \{s = (\sum i \in [0, x).\ i)\}$

è corretta e dunque anche la (†) lo è per la regola (pre).

Per quanto riguarda la terminazione del ciclo, scegliamo come funzione di terminazione la seguente:

$t : n - x.$

Dobbiamo allora dimostrare:

(1) $Inv \Rightarrow t \geq 0$;

(2) $\{Inv \wedge x < n \wedge t = V\}\ x, s := x+1, s+x\ \{t < V\}$

La (1) si dimostra facilmente:

$$
\begin{array}{ll}
 & s = (\sum i \in [0,x).\ i) \wedge 0 \leq x \leq n \\
\Rightarrow & \quad \{\ \textit{Sempl-}\wedge\ \} \\
 & x \leq n \\
\equiv & \quad \{\ \text{calcolo}\ \} \\
 & n - x \geq 0 \\
\equiv & \quad \{\ t = n - x\ \} \\
 & t \geq 0.
\end{array}
$$

Per quanto riguarda (2) abbiamo che la tripla

$\{n - x - 1 < V\}\ x, s := x+1, s+x\ \{n - x < V\}$

è un'istanza di (ass). Infine, osservando che $n - x = V \Rightarrow n - x - 1 < V$, otteniamo (2) per la regola (pre).

Esempio 4.14 Consideriamo il problema di calcolare quoziente e resto della divisione intera tra due numeri naturali. Osserviamo che, fissati x e y, q e r rappresentano, rispettivamente, il quoziente e il resto della divisione intera tra x e y se e solo se vale la proprietà:

$$x = q \cdot y + r \wedge 0 \leq r < y \tag{9}$$

Consideriamo allora la seguente proprietà:

$$x = q \cdot y + r \wedge 0 \leq r \tag{10}$$

ottenuta eliminando dalla congiunzione (9) il membro $r < y$. Intuitivamente, la proprietà (10) stabilisce che, in un generico stato, i valori di q e r sono *potenziali* valori per il quoziente ed il resto. Un comando iterativo che mantenga tale proprietà e che assicuri che, nello stato finale, il valore di r è minore del valore di y è allora una soluzione del problema.

In altre parole, sviluppiamo un comando iterativo del tipo

while $r \geq y$ **do** C **endw**

che utilizza la (10) come invariante.[3] A questo scopo, bisogna che il corpo C soddisfi l'ipotesi di invarianza, ovvero la seguente tripla:

$\{x = q \cdot y + r \wedge 0 \leq r \wedge r \geq y\}\ C\ \{x = q \cdot y + r \wedge 0 \leq r\}$.

Osserviamo che, assumendo $r \geq y$, si ha:

$$
\begin{array}{ll}
 & x = q \cdot y + r \\
\equiv & \quad \{\ \text{calcolo}\ \} \\
 & x = (q+1) \cdot y + (r - y) \\
\equiv & \quad \{\ \textbf{Ip:}\ r \geq y\ \} \\
 & x = (q+1) \cdot y + (r - y) \wedge 0 \leq r - y
\end{array}
$$

Poiché

$\{x = (q+1) \cdot y + (r - y) \wedge 0 \leq (r - y)\}\ q, r := q+1, r-y\ \{x = q \cdot y + r \wedge 0 \leq r\}$

è un'istanza di (ass), la precedente dimostrazione consente di concludere (grazie a (pre)) la correttezza della tripla

[3] Dal momento che $def(r \geq y) \equiv \mathbf{tt}$, nel seguito omettiamo di considerare $def(r \geq y)$.

$\{x = q \cdot y + r \wedge 0 \leq r \wedge r \geq y\}\ q, r := q + 1, r - y\ \{x = q \cdot y + r \wedge 0 \leq r\}$

Abbiamo dunque che, per il comando

while $r \geq y$ **do** $q, r := q + 1, r - y$ **endw**

la (10) è una buona invariante. Osserviamo inoltre che:

$$
\begin{array}{ll}
 & x = q \cdot y + r \wedge 0 \leq r \wedge \neg(r \geq y) \\
\equiv & \quad \{\ \neg(r \geq y) \equiv (r < y)\ \} \\
 & x = q \cdot y + r \wedge 0 \leq r < y)
\end{array}
$$

che corrisponde alla postcondizione desiderata. Per applicare la regola (*while*) dobbiamo tuttavia mostrare ancora che il ciclo termina, ovvero determinare una funzione di terminazione. Intuitivamente r rappresenta una buona funzione di terminazione, visto che il valore di r viene diminuito di y ad ogni iterazione. Formalmente dobbiamo mostrare che la (10) implica $r \geq 0$, che è immediato, ed inoltre dobbiamo mostrare la correttezza della tripla:

$\{x = q \cdot y + r \wedge 0 \leq r \wedge r \geq y \wedge r = V\}\ q, r := q + 1, r - y\ \{r < V\}$.

Possiamo appellarci ad (*ass*) concludendo che vale la tripla

$\{r - y < V\}\ q, r := q + 1, r - y\ \{r < V\}$

per poi dimostrare l'implicazione

$(x = q \cdot y + r \wedge 0 \leq r \wedge r \geq y \wedge r = V) \;\Rightarrow\; r - y < V$.

È però facile convincersi che la precedente implicazione non può essere dimostrata se non rafforzando la precondizione con l'ulteriore condizione $y > 0$. Dunque, la terminazione del ciclo può essere garantita solo a patto di rafforzare l'invariante in questo modo. Dal momento che il corpo del ciclo non modifica il valore di y, è chiaro che anche la nuova proprietà

$Inv : x = q \cdot y + r \wedge 0 \leq r \wedge 0 < y$

continua ad essere invariante per il ciclo.

Per quanto visto fino ad ora possiamo concludere, grazie alla regola (*while*), che vale la tripla

$\{Inv\}$ **while** $r \geq y$ **do** $q, r := q + 1, r - y$ **endw** $\{x = q \cdot y + r \wedge 0 \leq r \wedge r < y\}$.

Tuttavia, la precondizione iniziale del problema, ovvero $\{x \geq 0 \wedge y > 0\}$ non basta per garantire l'invariante nello stato iniziale. Dobbiamo allora premettere al ciclo stesso un comando di inizializzazione C_I che soddisfi la tripla

$\{x \geq 0 \wedge y > 0\}\ C_I\ \{Inv\}$.

È facile convincersi che tale comando può essere dato dall'assegnamento

$q, r := 0, x$.

Concludiamo allora che vale la tripla

$\{x \geq 0\ \wedge\ y > 0\}$
  $q, r := 0, x;$
  while $r \geq y$ **do**
    $q, r := q + 1, r - y$
  endw
$\{x = q \cdot y + r \wedge 0 \leq r < y\}$

4.8 Regole derivate

Elenchiamo in questo paragrafo ulteriori regole utili nella dimostrazione di formule di correttezza. Tali regole non sono essenziali, nel senso che la loro validità può essere dimostrata utilizzando le regole già date, ma semplificano le dimostrazioni di correttezza, in quanto consentono di "combinare" differenti formule di correttezza per uno stesso programma che siano state dimostrate separatamente.

La prima regola asserisce che, se un comando non modifica le variabili che occorrono libere in una asserzione, tale asserzione è invariante per il comando stesso. La notazione *change*(C) indica l'insieme delle variabili che, in C, compaiono a sinistra di un comando di assegnamento. Ad esempio $\mathit{change}(x, y := \ldots) = \{x, y\}$.

$$\frac{\{R\}\ C\ \{Q\}}{\{P \wedge R\}\ C\ \{P \wedge Q\}} \quad \text{se } \mathit{free}(P) \cap \mathit{change}(C) = \emptyset \qquad \text{(invariante)}$$

Le due regole seguenti sono utili per affrontare la dimostrazione di formule in cui le precondizioni e/o postcondizioni sono costituite da disgiunzioni o congiunzioni di asserzioni.

$$\frac{\{P_1\}\ C\ \{Q_1\} \qquad \{P_2\}\ C\ \{Q_2\}}{\{P_1 \wedge P_2\}\ C\ \{Q_1 \wedge Q_2\}} \quad (\mathit{cong})$$

$$\frac{\{P\}\ C\ \{Q\} \qquad \{R\}\ C\ \{Q\}}{\{P \vee R\}\ C\ \{Q\}} \quad (\mathit{disg})$$

4.9 Sequenze e aggiornamento selettivo

Vogliamo ora modificare il nostro linguaggio in modo da permettere l'uso delle *sequenze* (o *array*). Ricordiamo che una sequenza di elementi di un certo insieme $\mathcal{A}$ può essere rappresentata formalmente come una funzione da un intervallo $[a, b)$ di numeri naturali nell'insieme $\mathcal{A}$ in questione

$$\phi : [a, b) \rightarrow \mathcal{A}.$$

L'intervallo $[a, b)$ è detto *dominio* della sequenza. Dato un numero naturale $i \in [a, b)$, l'elemento $\phi(i)$ è detto elemento di posto i nella sequenza.

Supponiamo dunque di arricchire il linguaggio consentendo l'uso di identificatori di tipo sequenza (di booleani o di interi). Se v è un identificatore di tipo sequenza, indichiamo con la sintassi $v[E]$ l'operazione di selezione dell'elemento di v che occupa il posto dato dal valore dell'espressione E. La sintassi delle espressioni viene estesa come segue:

$$\texttt{Exp} ::= \texttt{Const} \mid \texttt{Ide} \mid \texttt{Ide}[\texttt{Exp}] \mid \texttt{(Exp)} \mid \texttt{Exp Op Exp} \mid \mathit{not}\ \texttt{Exp}.$$

Per indicare che una variabile v è di tipo "sequenza di elementi di tipo T", con dominio $[a, b)$, scriveremo:

$$v : \ \textbf{array}\ [a, b)\ \textbf{of}\ T,$$

dove T sarà **int** per sequenze di interi, e **bool** per sequenze di booleani. Con $\mathit{dom}(v)$ si indica il dominio $[a, b)$.

A questo punto dovrebbe essere chiaro che, se v è una sequenza di booleani e σ è uno stato, allora il valore di v nello stato σ è una funzione del tipo

$$\sigma(v) : \mathit{dom}(v) \longrightarrow \mathbb{B},$$

ed allo stesso modo, se v è una sequenza di interi, $\sigma(v)$ è una funzione $\sigma(v) : \mathit{dom}(v) \longrightarrow \mathbb{Z}$.

La definizione della semantica delle espressioni si estende in modo semplice aggiungendo:

$$\mathcal{E}(x[E]) = \mathcal{E}(x)(\mathcal{E}(E)), \qquad \text{se } \mathcal{E}(E) \in dom(x).$$

Si noti che non sempre la valutazione di una espressione $v[E]$ dà un valore: $v[E]$ è, infatti, definita solo in quegli stati in cui, oltre ad essere definita E, il valore di quest'ultima fa parte del dominio della sequenza stessa. Dunque si deve estendere anche la definizione di $def()$:

$$\boxed{def(v[E]) \quad = \quad def(E) \wedge E \in dom(v).}$$

Le operazioni del linguaggio si applicano solo a identificatori semplici o a singoli elementi di una sequenza (cioè espressioni del tipo $v[E]$), ma non a sequenze viste come entità atomiche. Per esempio, se x, y sono variabili semplici e a, b sono sequenze di interi con dominio $[0, 10)$, allora sono espressioni corrette:

$$3 * x + 5 \qquad\qquad a[2] * y + x \qquad\qquad a[2] \leq b[3],$$

mentre non sono possibili confronti o somme tra sequenze, come

$$a \leq b \qquad\qquad a + b.$$

Allo stesso modo il comando di assegnamento è esteso in modo da poter avere nella parte sinistra solo singoli elementi di sequenze, ma non sequenze. Formalmente, cambia la definizione di `Ide_List` come segue:

`Ide_List` ::= `Ide` | `Ide[Exp]` | `Ide, Ide_List` | `Ide[Exp], Ide_List`

Consideriamo ora un comando di assegnamento del tipo

$$v[E] := E',$$

che chiameremo *aggiornamento selettivo*. L'effetto di tale comando è di transire, a partire da uno stato σ, allo stato $\sigma[{}^{v[{}^{\mathcal{E}(E',\sigma)}/_{\mathcal{E}(E,\sigma)}]}/_{v}]$, a patto che le due espressioni in gioco siano definite in σ e che il valore di E in σ stia nel dominio di v. La generalizzazione dell'assioma (ass) per il comando di aggiornamento selettivo precedente è la seguente:

$$\boxed{\begin{array}{r}\{def(E) \wedge def(E') \wedge E \in dom(v) \wedge P^{v'}_{v}\}\ v[E] := E'\ \{P\} \qquad \text{dove } v' = v[{}^{E'}/_{E}] \\[1em] (agg - sel)\end{array}}$$

Vediamo alcuni esempi di applicazione di tale regola.

Esempio 4.15 Verificare la tripla:

$$\{k \in dom(v) \wedge (\forall i : i \in dom(v) \wedge i \neq k.\ v[i] > 0)\}\ v[k] := 3\ \{(\forall i \in dom(v).\ v[i] > 0)\}$$

L'applicazione di ($agg - sel$) in questo esempio ci garantisce la correttezza della seguente tripla (in cui omettiamo $def(k)$ e $def(3)$):

$$\{k \in dom(v) \wedge (\forall i \in dom(v').\ v'[i] > 0)\}\ v[k] := 3\ \{(\forall i \in dom(v).\ v[i] > 0)\},$$

dove $v' = v[{}^{3}/_{k}]$. Abbiamo:

$$\begin{array}{ll}
 & k \in dom(v) \wedge (\forall i \in dom(v').\ v'[i] > 0) \\
\equiv & \qquad \{\ \textbf{Ip:}\ k \in dom(v)\ \} \\
 & (\forall i :\ i \in dom(v').\ v'[i] > 0) \\
\equiv & \qquad \{\ \textbf{Ip:}\ v' = v[{}^{3}/_{k}],\ dom(v) = dom(v')\ \} \\
 & (\forall i :\ i \in dom(v).\ v[{}^{3}/_{k}][i] > 0)
\end{array}$$

$\equiv$ $\quad\{$ **Ip:** $k \in dom(v)$ $\}$
$(\forall i: \ i \in dom(v) \wedge i \neq k.\ v[{}^3/_k][i] > 0) \wedge v[{}^3/_k][k] > 0$
$\equiv$ $\quad\{ v[{}^3/_k][k] = 3, 3 > 0 \}$
$(\forall i: \ i \in dom(v) \wedge i \neq k.\ v[{}^3/_k][i] > 0)$
$\equiv$ $\quad\{ (\forall i: \ i \in dom(v) \wedge i \neq k.\ v[{}^3/_k][i] = v[i]) \}$
$(\forall i: \ i \in dom(v) \wedge i \neq k.\ v[i] > 0)$

Abbiamo dunque dimostrato

$k \in dom(v) \wedge (\forall i \in dom(v).\ v[i] > 0)$
$\Rightarrow$ $\quad\{$ **Ip:** $v' = v[{}^3/_k]$ $\}$
$k \in dom(v) \wedge (\forall i \in dom(v').\ v'[i] > 0)$

che ci permette di concludere che la tripla è corretta, grazie alla regola (*pre*).

Nell'esempio abbiamo utilizzato la proprietà 2.2 dell'operazione di aggiornamento di funzione, denotata da $f[{}^c/_i]$.

5 Programmi annotati

Le dimostrazioni di correttezza di programmi sono piuttosto lunghe e noiose, essendo costituite spesso da numerosi e piccoli passi di dimostrazione. Un modo alternativo per organizzare una dimostrazione di correttezza consiste nel suddividerla in alcuni passi principali, guidati dalla struttura del programma. La dimostrazione può allora essere presentata attraverso un *programma annotato* nel quale le varie porzioni del programma sono inframezzate da asserzioni che descrivono lo stato dell'esecuzione e che rappresentano anche i fatti essenziali da dimostrare per ottenere la correttezza del programma stesso.

Una domanda che sorge spontanea è: "Quanto densamente un programma dovrebbe essere annotato con asserzioni?". L'idea è che dovremmo poter guardare al frammento di programma tra due asserzioni consecutive come ad un entità atomica, facilmente comprensibile senza bisogno di un'ulteriore suddivisione. Certo, semplici sequenze di assegnamenti parrebbero sufficientemente elementari da poter essere facilmente comprensibili, ma anche queste ci riservano alcune "piccole sorprese".

Ad esempio, il programma $z := x;\ x := y;\ y := z$ scambia i valori delle variabili x e y, mentre nel programma $x := y;\ y := x$ il secondo assegnamento è inutile, ma per capirlo è necessario un momento di riflessione ...

I comandi condizionali tipicamente non aggiungono complicazioni concettuali. Dobbiamo semplicemente pensare a due alternative disgiunte e questo non è, in genere, più complicato che pensare a due programmi eseguiti in sequenza. Ovviamente anche in questo caso ci sono delle eccezioni e dei limiti. Per esempio, non è immediato capire che, nel seguente programma, la condizione più interna non è necessaria.

if $x < y$
  then if $x < z$
        then ...
        else if $z < y$ **then** ... **fi**
      fi
  else ...
fi

La situazione cambia decisamente nel caso dei cicli. Anche cicli con un corpo semplice sono, talvolta, complicati da capire se non si ha a disposizione una documentazione aggiuntiva. Si consideri, per esempio, il seguente programma, dove si suppone che inizialmente $x \geq 0$:

$a, b, y := 0, 0, 0;$
while $y \leq x$ **do**
      $y := y + 3 * a + 3 * b + 1;$
      $a := a + 2 * b + 1;$
      $b := b + 1$
endw.

Non è certo evidente, a prima vista, cosa calcoli il programma. La risposta è che calcola, anche se in un modo primitivo, la radice cubica intera di x, ovvero, $\lfloor \sqrt[3]{x} \rfloor$. Più precisamente, quando il ciclo termina abbiamo che $(b-1)^3 \leq x < b^3$. Durante il ciclo $a = b^2$ e $y = b^3$. Per mantenere "aggiornati" i valori di a e y il programma usa le formule algebriche: $(z+1)^2 = z^2 + 2z + 1$ e $(z+1)^3 = z^3 + 3z^2 + 3z + 1$.

Queste spiegazioni appaiono necessarie per la comprensione del programma ed il modo migliore per inserirle nel programma stesso è fornire l'invariante del ciclo, che è semplicemente

$$a = b^2 \ \wedge \ y = b^3 \ \wedge \ (b-1)^3 \leq x.$$

Anche se non è buona abitudine generalizzare a partire da un singolo esempio, questa situazione è piuttosto tipica. Molti cicli sono incomprensibili senza informazioni addizionali. Asseriamo che un ciclo può essere capito una volta che abbiamo fornito anche l'invariante, dunque l'invariante di un ciclo dovrebbe essere sempre indicato.

Dunque, non è facile dare una regola precisa e generale. Il buonsenso e l'esperienza devono guidarci nell'individuazione delle asserzioni che rappresentino passi significativi, ma non troppo complicati di dimostrazione. Vediamo alcuni esempi.

Esempio 5.1 Consideriamo la tripla:

$$\begin{array}{l} \{x = 10 \ \wedge \ y = 20\} \\ \quad v, w := z \ mod \ x, z \ mod \ y; \\ \quad \textbf{if } v \geq w \ \textbf{ then skip } \ \textbf{ else } v := w \ \textbf{fi} \\ \{v = (z \ mod \ x) \ \textbf{max} \ (z \ mod \ y)\} \end{array}$$

Un'organizzazione della dimostrazione di questa tripla può essere rappresentata dal seguente programma annotato:

$$\begin{array}{l} \{x = 10 \ \wedge \ y = 20\} \\ \{x \neq 0 \wedge y \neq 0\} \\ \quad v, w := z \ mod \ x, z \ mod \ y; \\ \{v = z \ mod \ x \wedge w = z \ mod \ y\} \\ \quad \textbf{if } v \geq w \ \textbf{ then skip } \ \textbf{ else } v := w \ \textbf{fi} \\ \{v = (z \ mod \ x) \ \textbf{max} \ (z \ mod \ y)\} \end{array}$$

che rappresenta i seguenti fatti da dimostrare:

(i) $(x = 10 \ \wedge \ y = 20) \ \Rightarrow \ (x \neq 0 \wedge y \neq 0)$

(ii) $\{x \neq 0 \wedge y \neq 0\} \ v, w := z \ mod \ x, z \ mod \ y \ \{v = z \ mod \ x \wedge w = z \ mod \ y\}$

(iii) $$\begin{array}{l} \{v = z \ mod \ x \wedge w = z \ mod \ y\} \\ \quad \textbf{if } v \geq w \ \textbf{ then skip } \ \textbf{ else } \mathrm{v} := \mathrm{w} \ \textbf{fi} \\ \{v = (z \ mod \ x) \ \textbf{max} \ (z \ mod \ y)\} \end{array}$$

Si noti che

$$\begin{array}{l} \{x = 10 \ \wedge \ y = 20\} \\ \{x \neq 0 \wedge y \neq 0\} \\ \quad v, w := z \ mod \ x, z \ mod \ y; \\ \{v = z \ mod \ x \wedge w = z \ mod \ y\} \end{array}$$

rappresenta in modo compatto il fatto che la tripla:

$$\{x = 10 \ \wedge \ y = 20\} \ v, w := z \ mod \ x, z \ mod \ y \ \{v = z \ mod \ x \wedge w = z \ mod \ y\}$$

può essere dimostrata utilizzando (pre) e l'istanza di (ass) data da

$$\{x \neq 0 \wedge y \neq 0\} \ v, w := z \ mod \ x, z \ mod \ y \ \{v = z \ mod \ x \wedge w = z \ mod \ y\}.$$

Per quanto riguarda (iii) un'ulteriore raffinamento è dato dal seguente programma annotato:

$$\begin{array}{l} \{P\} \\ \quad \textbf{if } v \geq w \\ \quad\quad \textbf{then } \{P \wedge v \geq w\} \ \textbf{skip} \ \ \{Q\} \\ \quad\quad \textbf{else } \{P \wedge v < w\} \ v := w \ \{Q\} \\ \quad \textbf{fi} \\ \{Q\} \end{array}$$

dove

$P \equiv v = z \ mod \ x \wedge w = z \ mod \ y$, e

$Q \equiv v = (z \ mod \ x) \ \textbf{max} \ (z \ mod \ y)$.

Il prossimo esempio mette in luce la forma di programmi annotati che coinvolgono comandi ripetitivi.

Esempio 5.2 Si consideri il programma annotato per il calcolo del quoziente e resto della divisione intera di numeri naturali (si veda l'esempio 4.14).

$\{x \geq 0 \ \wedge \ y > 0\}$
$\quad q, r := 0, x;$
$\{\mathsf{inv} : P\} \ \{\mathsf{ft} : r\}$
$\quad$ **while** $r \geq y$ **do**
$\quad \{P \wedge r \geq y\}$
$\qquad q, r := q + 1, r - y$
$\quad$ **endw**
$\{P \wedge r < y\}$
$\{q \cdot y = x \wedge 0 \leq r < y\}$

dove, come visto nell'esempio 4.14,

$$P \equiv q \cdot y + r = x \ \wedge \ r \geq 0 \wedge y > 0.$$

Si noti l'uso della notazione $\{\mathsf{inv} : P\} \ \{\mathsf{ft} : r\}$ ad indicare che P è l'invariante e r la funzione di terminazione del comando iterativo che segue. Anche in questo esempio, due asserzioni adiacenti del tipo $\{P\} \ \{Q\}$ stanno ad indicare che vale l'implicazione $P \Rightarrow Q$: nell'esempio le ultime due asserzioni $\{P \wedge 0 \leq r < y\}$ $\{q \cdot y = x \wedge 0 \leq r < y\}$ rappresentano la premessa $\mathit{Inv} \wedge \neg E \ \Rightarrow \ Q$ della regola (while).

Generalizzando l'esempio, la correttezza di una tripla del tipo

$$\{P\} \ C_I; \textbf{while } E \textbf{ do } C \textbf{ endw } \{Q\}$$

verrà rappresentata dal seguente programma annotato:

$\{P\}$
$\quad C_I;$
$\{\mathsf{inv} : \mathit{Inv}\}\{\mathsf{ft} : t\}$
$\quad$ **while** E **do**
$\qquad \{\mathit{Inv} \wedge E\}$
$\qquad C$
$\quad$ **endw**
$\{\mathit{Inv} \wedge \neg E\}$
$\{Q\}$

che rappresenta i seguenti fatti da dimostrare:

$$\{P\} \ C_I \ \{\mathit{Inv} \wedge \mathit{def}(E)\}$$

$$\{\mathit{Inv} \wedge E\} \ C \ \{\mathit{Inv} \wedge \mathit{def}(E)\}$$

$$\mathit{Inv} \ \Rightarrow \ t \geq 0$$

$$\{\mathit{Inv} \wedge E \wedge t = V\} \ C \ \{t < V\}$$

$$\mathit{Inv} \wedge \neg E \ \Rightarrow \ Q$$

6 Problemi e specifiche

Un problema viene normalmente descritto in modo informale specificando i dati iniziali del problema e le proprietà che tali dati soddisfano, ed indicando quali risultati finali si vorrebbero ottenere. La forma tipica è la seguente

"Dati $\langle$*dati iniziali*$\rangle$ tali che $\langle$*proprietà dei dati*$\rangle$, calcolare $\langle$*risultati*$\rangle$".

Se il problema deve essere risolto con un calcolatore, utilizzando ad esempio il nostro linguaggio di programmazione, inizialmente i dati saranno contenuti in variabili. Quindi i dati iniziali e le loro proprietà sono esprimibili tramite un'asserzione sullo stato (iniziale). Allo stesso modo, al termine dell'elaborazione, i risultati saranno contenuti in variabili e quindi, ancora, i risultati che si vogliono ottenere possono essere espressi mediante un'asserzione sullo stato (finale). Pertanto, come già accennato, un problema di programmazione può essere specificato formalmente, in modo naturale, attraverso una tripla di Hoare

$$\{Q\}\ C\ \{R\},$$

nella quale sono note la precondizione Q e la post-condizione R. Risolvere il problema significa determinare un comando C che verifichi la tripla suddetta. In questo paragrafo illustreremo, tramite una serie di esempi, come sia possibile formalizzare un problema formulato in linguaggio naturale, determinando le asserzioni corrette che lo specificano.

Esempio 6.1

Problema

Determinare il massimo tra i valori delle variabili naturali x e y.

Specifica

Una prima formulazione della specifica del problema potrebbe essere la seguente:

$$\{x \geq 0\ \wedge\ y \geq 0\}\ C\ \{z = x\ \mathbf{max}\ y\} \qquad (\dagger)$$

La precondizione della specifica ($\dagger$) data mette in evidenza il fatto che x e y sono variabili a valori naturali e la postcondizione il fatto che la variabile z nello stato finale assume il massimo tra i valori di x e y. È facile convincersi che il comando

if $x \geq y$ **then** $z := x$ **else** $z := y$ **fi**

è una possibile soluzione del problema.

Si noti, tuttavia, che la specifica ($\dagger$) non dice nulla riguardo al valore delle variabili x e y nello stato finale. Dunque, una soluzione alternativa (seppure non intuitiva) al problema precedente è data dal comando seguente:

$$x, y, z := 2, 3, 3.$$

La descrizione informale del problema lascia invece intendere che il valore delle variabili x e y deve rimanere inalterato a seguito dell'esecuzione del comando, ovvero che x e y devono essere considerate *costanti* del problema stesso. Se indichiamo rispettivamente con X e Y i valori delle variabili x e y nello stato iniziale, quanto detto può essere espresso formalmente asserendo che la congiunzione $x = X \wedge y = Y$ deve essere *invariante* per il comando C. Alla luce di queste considerazioni, una specifica più appropriata del problema è la seguente:

$$\{x = X\ \wedge\ y = Y\ \wedge\ X \geq 0\ \wedge\ Y \geq 0\}\ C\ \{z = x\ \mathbf{max}\ y\ \wedge\ x = X\ \wedge\ y = Y\} \qquad (\dagger\dagger)$$

Si noti che, con tale specifica, il comando

if $x \geq y$
  then $z := x$
  else $z := y$
fi

è ancora una soluzione del problema, mentre non lo è più, in generale, il comando

$x, y, z := 2, 3, 3.$

Nella specifica (††), la presenza di $x = X \ \wedge \ y = Y$ sia nella precondizione che nella postcondizione sta a significare che i valori, nello stato finale, delle variabili di programma x e y devono rimanere quelli che x e y hanno nello stato iniziale. Rilasciando questa condizione, una ulteriore possibile specifica dello stesso problema è:

$$\{x = X \ \wedge \ y = Y \ \wedge \ X \geq 0 \ \wedge \ Y \geq 0\} \ C \ \{z = X \ \mathbf{max} \ Y\} \qquad (†††)$$

Qui la specifica indica solo il fatto che la variabile z nello stato finale deve assumere come valore il massimo tra i valori di x e y nello stato iniziale, ma nulla si dice riguardo ai valori di x e y nello stato finale. Dunque questa specifica è più debole rispetto a (††), ovvero l'insieme dei programmi che la soddisfano è più ampio. Per esempio il seguente comando soddisfa (†††), ma non (††).

if $x \geq y$
  then $z := x$
  else $z := y$
fi;
$x, y := 2, 3$

È fondamentale osservare che X e Y rappresentano generici valori naturali e in quanto tali non possono essere utilizzati all'interno del comando che risolve il problema.

Come ulteriore esempio, riconsideriamo i problema del calcolo di quoziente e resto della divisione intera (cfr. Esempio 4.14).

Esempio 6.2

Problema
Determinare il quoziente ed il resto della divisione intera tra x e y, con x, y naturali e y non nullo.

Anche in questo caso una specifica che non si riferisca ai valori delle variabili nello stato iniziale:

$$\{x \geq 0 \ \wedge \ y > 0\} \ C \ \{x = q \cdot y + r \wedge 0 \leq r < y\}$$

è evidentemente soddisfatta dal comando

$x, y, q, r := 7, 5, 1, 2$

La specifica informale, anche in questo caso, lascia intendere che i valori di x, y sono dati e devono rimanere gli stessi anche nello stato finale. Dunque, una specifica corretta del problema può essere data utilizzando le variabili di specifica come segue:

$$\{x = X \wedge y = Y \wedge X \geq 0 \ \wedge \ Y > 0\} \ C \ \{x = X \wedge y = Y \wedge x = q \cdot y + r \wedge 0 \leq r < y\}$$

Variabili di Specifica

Nel seguito gli identificatori che servono per rappresentare valori generici (come X e Y nell'esempio precedente) verranno chiamati *variabili di specifica* per distinguerli dalle *variabili di programma*, come x e y nell'esempio precedente. Ovviamente le variabili di specifica rappresentano valori generici che non possono cambiare durante l'esecuzione dei comandi, a differenza delle variabili di programma. Useremo lettere *maiuscole* per indicare *variabili di specifica* e *lettere minuscole* per indicare *variabili di programma.*

Una ulteriore situazione nella quale è indispensabile usare variabili di specifica è illustrata dal seguente esempio.

Esempio 6.3

Problema

Raddoppiare il valore della variabile naturale x.

La formalizzazione della specifica di problemi come questo, dove il valore di una variabile nello stato finale è funzione del valore della stessa nello stato iniziale, non può prescindere dall'uso di variabili di specifica. Infatti, la specifica ingenua data da

$$\{x \geq 0\}\ C\ \{x = 2x\}$$

non rappresenta correttamente il problema dato. Ricordiamo infatti che un identificatore di variabile x in una asserzione sta ad indicare il valore di x in uno stato. Dunque, uno stato σ soddisfa la postcondizione precedente se e soltanto se il valore di x in σ è 0! Di conseguenza una soluzione del problema così specificato è un comando del tipo $x := 0$, o un qualunque altro comando il cui effetto sia quello di azzerare il valore della variabile x.

Una corretta specifica del problema dato si ottiene invece introducendo una variabile di specifica, ad esempio A, che rappresenti il valore di x nello stato iniziale. Formalmente:

$$\{x = A\ \wedge\ A \geq 0\}\ C\ \{x = 2A\}.$$

Con tale specifica una corretta soluzione del problema è, tra le altre, quella data dal seguente comando:

$$x := 2 * x.$$

Si noti che un comando del tipo

$$x := 2 * A.$$

non risolve il problema dato, o meglio, l'espressione $2 * A$ non è lecita, dal momento che A è una variabile di specifica che rappresenta il *generico* valore di x nello stato iniziale.

L'introduzione delle variabili di specifica nelle asserzioni richiede di riconsiderare la definizione 4.2. In primo luogo, le variabili di specifica eventualmente presenti in un'asserzione P non devono essere viste come variabili libere in P. Le variabili libere di un'asserzione sono tutte e sole le eventuali variabili di programma in esse presenti. Poiché le variabili di specifica indicano generici valori, un'uguaglianza del tipo $x = A$ è soddisfatta in un qualunque stato in cui alla variabile di programma x è associato un valore.

Esempio 6.4 Sia P l'asserzione $\{x = A \wedge y = B \wedge z = A\ \mathbf{max}\ B\}$ e sia σ uno stato tale che $\sigma(x) = 3$, $\sigma(y) = 15$ e $\sigma(z) = 15$. Formalmente abbiamo:

$$
\begin{array}{ll}
 & \sigma\ \models\ P \\
\equiv & \qquad \{\text{ definizione 4.2 }\} \\
 & 3 = A\ \wedge\ 15 = B\ \wedge\ 15 = A\ \mathbf{max}\ B \\
\equiv & \qquad \{\text{ Leibniz }\} \\
 & 3 = A\ \wedge\ 15 = B\ \wedge\ 15 = 3\ \mathbf{max}\ 15 \\
\equiv & \qquad \{\ 15 = 3\ \mathbf{max}\ 15, \text{ Unità }\} \\
 & 3 = A\ \wedge\ 15 = B
\end{array}
$$

Pur non avendo ridotto $\sigma\ \models\ P$ a $\mathbf{tt}$, siamo autorizzati a dire che lo stato σ soddisfa P poiché A e B, che sono variabili di specifica, rappresentano valori generici e quindi, in particolare, i valori 3 e 15 rispettivamente.

Esempio 6.5 Sia P l'asserzione $x = A \wedge y = A$ che, intuitivamente, rappresenta un generico stato in cui le variabili di programma x e y hanno lo stesso valore, denotato dalla variabile di specifica A. È evidente che $\sigma\ \models\ P$ se e solo se $\sigma(x) = \sigma(y)$. Consideriamo ad esempio σ tale che $\sigma(x) = 10$ e $\sigma(y) = 20$.

$$
\begin{array}{ll}
 & \sigma \models P \\
\equiv & \quad \{ \text{ definizione 4.2 } \} \\
 & 10 = A \ \wedge \ 20 = A \\
\equiv & \quad \{ \text{ Leibniz } \} \\
 & 10 = A \ \wedge \ 10 = 20 \\
\equiv & \quad \{ \neg(10 = 20), \text{ Zero } \} \\
 & \mathbf{ff}
\end{array}
$$

Convenzioni

Per semplificare la notazione utilizzata, adotteremo alcune convenzioni nella scrittura di programmi e asserzioni. Ricordiamo innanzitutto che le *variabili di specifica* sono rappresentate da lettere maiuscole, mentre le *variabili di programma* sono rappresentate da lettere minuscole.

Inoltre, per evitare di appesantire troppo le specifiche formali, le variabili di programma il cui valore deve rimanere *costante* nell'ambito di un programma verranno indicate esplicitamente nel testo. Ad esempio, in riferimento agli esempi 6.1 e 6.2 daremo specifiche semplificate del tipo:

$\{x \geq 0 \ \wedge \ y \geq 0\} \ C \ \{z = x \ \mathbf{max} \ y\}$, con x, y costanti;

$\{x \geq 0 \ \wedge \ y > 0\} \ C \ \{x = q \cdot y + r \wedge 0 \leq r < y\}$, con x, y costanti;

intendendo con ciò che il valore delle variabili di programma x, y deve rimanere inalterato. In questo modo evitiamo di dover indicare esplicitamente una condizione del tipo $x = X \ \wedge \ y = Y$ sia nella precondizione che nella postcondizione.

Un'ultima convenzione riguarda l'uso di variabili di specifica che corrispondono a sequenze (array). Consideriamo il seguente esempio.

Esempio 6.6

Problema

Incrementare di 1 tutti gli elementi dell'array di interi a con dominio $[0, n)$.

È chiaro che, come nell'esempio 6.3, abbiamo bisogno di introdurre una variabile di specifica che rappresenti il valore iniziale dell'array a. Indicando con V tale variabile di specifica, possiamo utilizzare, nella precondizione e nella precondizione della specifica formale, una formula del tipo

$$(\forall k \in [0, n).a[k] = V[k]).$$

Per semplificare la notazione, abbrevieremo questa formula con l'uguaglianza

$$a = V,$$

ottenendo così la specifica formale

$\{dom(a) = [0, n) \ \wedge \ a = V\} \ C \ \{(\forall k \in [0, n). \ a[k] = V[k] + 1)\}$, con n costante.

Diamo di seguito alcuni esempi di specifiche che utilizzano le convenzioni adottate.

- **Problema 1** Scambiare i valori di due variabili x, y.

Specifica $\{x = A \ \wedge \ y = B\} \ C \ \{x = B \ \wedge \ y = A\}$

- **Problema 2** Calcolare la parte intera della radice quadrata di un numero naturale.

Specifica $\{n \geq 0\} \ C \ \{x = \lfloor \sqrt{n} \rfloor\}$ con n costante

- **Problema 3** Azzerare gli elementi di indice pari di un array.

Specifica $\{a = V\} \ C \ \{(\forall k \in dom(a). \ (pari(k) \ \wedge \ a[k] = 0) \vee (\neg pari(k) \ \wedge \ a[k] = V[k]))\}$

con $pari(x) \equiv (x \ mod \ 2 = 0)$

- **Problema 4** Invertire un array.

Specifica $\{dom(a) = [0, n) \ \wedge \ a = V\} \ C \ \{\forall k \in [0, n \ div \ 2).\ a[k] = V[n - k - 1]\}$

7 Schemi di programma

In questa sezione presentiamo alcuni programmi basati su cicli iterativi che consentono di risolvere molti problemi pratici. Si parla di *schemi* di programma in quanto la loro correttezza viene dimostrata rispetto ad alcuni parametri che vanno istanziati di volta in volta. Gli schemi prevedono delle condizioni di applicabilità che tali parametri devono soddisfare per garantire la correttezza delle loro istanziazioni. Per ogni schema verrà presentato prima un esempio e poi la sua generalizzazione.

7.1 Ricerca lineare certa

Si tratta del problema di determinare il minimo elemento di un intervallo dato che soddisfa una certa proprietà, sapendo che esiste almeno un elemento nell'intervallo che soddisfa tale proprietà (ipotesi di certezza). Vediamo un esempio.

Esempio 7.1 Dato un vettore di numeri naturali a con dominio $[0, n)$, che contiene almeno un elemento minore di 100, determinare il minimo indice x nel dominio del vettore tale che $a[x] < 100$.

Diamo di seguito alcune specifiche formali del problema. Nelle specifiche che seguono, a, n sono costanti e I è una variabile di specifica.

$$\{(\exists i \in [0, n).\ a[i] < 100)\}\ RLC\ \{x = \min\{j \in [0, n)\ \mid\ a[j] < 100\}\}$$

$$\{I \in [0, n) \wedge a[I] < 100\}\ RLC\ \{x = \min\{j \in [0, n)\ \mid\ a[j] < 100\}\}$$

$$\{I \in [0, n) \wedge a[I] < 100\}\ RLC\ \{x \in [0, n)\ \wedge\ (\forall j \in [0, x).\ a[j] \geq 100)\ \wedge\ a[x] < 100\}$$

Sviluppiamo il comando RLC a partire dall'ultima specifica data. Osserviamo innanzitutto:

$$\begin{array}{ll} & x \in [0, n)\ \wedge\ (\forall j \in [0, x).\ a[j] \geq 100)\ \wedge\ a[x] < 100 \\ \Leftarrow & \qquad \{\ \mathbf{Ip:}\ I \in [0, n) \wedge a[I] < 100\ \} \\ & x \in [0, I]\ \wedge\ (\forall j \in [0, x).\ a[j] \geq 100)\ \wedge\ a[x] < 100 \end{array}$$

Siano dunque:

pre: $I \in [0, n) \wedge a[I] < 100$

post: $x \in [0, I]\ \wedge\ (\forall j \in [0, x).\ a[j] \geq 100)\ \wedge\ a[x] < 100$

e la seguente specifica del problema:

$$\{pre\}\ RLC\ \{post\}$$

Dal momento che l'array in questione deve rimanere costante e che, dunque, l'ipotesi *pre* è invariante per il problema, nel seguito ometteremo di esplicitarla nelle triple, ma la utilizzeremo, quando necessario, nei calcoli.

La determinazione dell'invariante e della guardia del ciclo candidate a risolvere il problema, avviene manipolando l'asserzione *post* in modo da individuare *Inv* e G tali che $Inv \wedge \neg G \Rightarrow post$. In questo caso possiamo scegliere:

$$Inv \equiv x \in [0, I]\ \wedge\ (\forall j \in [0, x).\ a[j] \geq 100)$$

$$G \equiv a[x] \geq 100$$

Sviluppiamo un comando C che soddisfi la tripla (invarianza del teorema di iterazione finita)

$$\{Inv \wedge G\}\ C\ \{Inv \wedge def(G)\}$$

Osserviamo:

$$
\begin{array}{ll}
 & \mathit{Inv} \wedge G \\
\equiv & \quad \{ \text{ definizione di } \mathit{Inv} \text{ e } G \ \} \\
 & x \in [0,I] \ \wedge \ (\forall j \in [0,x).\ a[j] \geq 100) \ \wedge \ a[x] \geq 100 \\
\Rightarrow & \quad \{ \ I \in [0,n) \ \wedge \ a[I] < 100 \ \} \\
 & x \in [0,I] \ \wedge \ (\forall j \in [0,x).\ a[j] \geq 100) \ \wedge \ a[x] \geq 100 \ \wedge \ x \neq I \\
\equiv & \quad \{ \ (x \in [0,I] \ \wedge \ x \neq I) \equiv x \in [0,I) \ \} \\
 & x \in [0,I) \ \wedge \ (\forall j \in [0,x).\ a[j] \geq 100) \ \wedge \ a[x] \geq 100 \\
\equiv & \quad \{ \text{ Intervallo } \} \\
 & x \in [0,I) \ \wedge \ (\forall j \in [0,x+1).\ a[j] \geq 100) \\
\Rightarrow & \quad \{ \ x \in [0,I) \Rightarrow x+1 \in [0,I] \ \} \\
 & x+1 \in [0,I] \ \wedge \ (\forall j \in [0,x+1).\ a[j] \geq 100) \\
\equiv & \quad \{ \text{ Sostituzione } \} \\
 & \mathit{Inv}_x^{x+1}
\end{array}
$$

Dunque da $\mathit{Inv} \wedge G$ e dalle ipotesi su I possiamo concludere che il comando $x := x + 1$ mantiene la proprietà Inv. Rimane da verificare che vale la tripla

$$\{\mathit{Inv} \wedge G\}\ x := x + 1\ \{\mathit{def}(G)\}$$

ovvero

$$\{\mathit{Inv} \wedge G\}\ x := x + 1\ \{x \in [0, n)\}.$$

Grazie ad (ass) vale

$$\{x + 1 \in [0, n)\}\ x := x + 1\ \{x \in [0, n)\}.$$

ed inoltre:

$$
\begin{array}{ll}
 & x+1 \in [0,n) \\
\equiv & \quad \{ \ \textbf{Ip:}\ x \in [0,I),\ I \in [0,n) \ \} \\
 & \mathbf{tt}
\end{array}
$$

Si noti l'uso dell'ipotesi $x \in [0, I)$, conseguenza di $\mathit{Inv} \wedge G$ come mostrato nella dimostrazione precedente.

Possiamo a questo punto stabilire che il programma RLC è del tipo

$$
\begin{array}{l}
\{\mathit{pre}\} \\
\quad C'; \\
\{\mathsf{inv} : \mathit{Inv}\}\ \{\mathsf{ft} :?\} \\
\quad \textbf{while } a[x] \geq 100 \textbf{ do} \\
\qquad x := x + 1 \\
\quad \textbf{endw} \\
\{\mathit{post}\}
\end{array}
$$

Per quanto riguarda la funzione di terminazione, osserviamo che il valore di x cresce ad ogni iterazione e quindi t può essere in funzione di $-x$. Inoltre $x \in [0, I]$, parte dell'invariante, garantisce che che la quantità $I - x$ è non negativa in tutti gli stati che soddisfano Inv. Concludiamo che una buona funzione di terminazione è:

$$t : I - x.$$

La dimostrazione formale delle ipotesi di terminazione e progresso è lasciata per esercizio.

Rimane infine da determinare un comando C' che soddisfi la tripla

$$\{\mathit{pre}\}\ C'\ \{\mathit{Inv} \wedge \mathit{def}(G)\}.$$

$$
\begin{array}{ll}
& \mathit{Inv} \wedge \mathit{def}(G) \\
\equiv & \quad \{ \text{ definizione di } \mathit{Inv} \text{ e } G, \text{ e } \mathit{def} \; \} \\
& x \in [0, I] \;\wedge\; (\forall j \in [0, x).\; a[j] \geq 100) \;\wedge\; x \in [0, n) \\
\equiv & \quad \{ \; \mathbf{Ip:}\; x = 0 \; \} \\
& 0 \in [0, I] \;\wedge\; (\forall j \in [0, 0).\; a[j] \geq 100) \;\wedge\; 0 \in [0, n) \\
\equiv & \quad \{ \; [0, 0) \text{ vuoto } \} \\
& 0 \in [0, I] \;\wedge\; 0 \in [0, n) \\
\equiv & \quad \{ \; I \in [0, n), \text{ dunque } [0, n) \text{ non vuoto } \} \\
& \mathbf{tt}.
\end{array}
$$

Concludiamo che $x := 0$ è il comando C' cercato.

Il procedimento sviluppato nell'esempio può essere facilmente generalizzato al caso di ricerca *certa* del minimo elemento di un intervallo $[a, b)$ che soddisfa una data proprietà G (esprimibile come espressione booleana del linguaggio di programmazione). L'unica osservazione importante riguarda il fatto che, nel caso generale, dobbiamo assicurare la definizione della guardia dopo ogni iterazione. Sia dunque G l'espressione booleana del linguaggio, che esprime la proprietà desiderata per il generico elemento $a[x]$ del vettore. La definizione di G durante l'esecuzione del programma può essere garantita mostrando semplicemente l'implicazione

$$x \in [a, b) \Rightarrow \mathit{def}(G),$$

anziché le più deboli:

$$\{\mathit{pre}\}\; C' \;\{\mathit{def}(G)\} \qquad \text{e} \qquad \{\mathit{Inv}\}\; C' \;\{\mathit{def}(G)\}.$$

Con questa osservazione, la generalizzazione dell'esempio porta al seguente schema di programma:

<u>**Ricerca lineare certa**</u>

$$
\begin{array}{l}
\{\mathit{pre}\} \\
\quad x := a; \\
\{\mathsf{inv} : \mathit{Inv}\} \; \{\mathsf{ft} : I - x\} \\
\quad \mathbf{while}\ \mathit{not}\ G\ \mathbf{do} \\
\qquad x := x + 1 \\
\quad \mathbf{endw} \\
\{\mathit{post}\}
\end{array}
$$

dove

pre: $I \in [a, b) \wedge G^I_x \wedge (x \in [a, b) \Rightarrow \mathit{def}(G))$

post: $x \in [a, b) \;\wedge\; (\forall j \in [a, x).\; \neg G^j_x) \;\wedge\; G$

Inv: $x \in [a, I] \;\wedge\; (\forall j \in [a, x).\; \neg G^j_x)$

7.2 Ricerca lineare incerta

Si tratta del problema di determinare, se esiste, il minimo elemento di un intervallo dato che soddisfa una certa proprietà. Come nel caso della ricerca lineare certa, è conveniente assumere che nell'intervallo in questione la proprietà sia definita. Il fatto che un elemento che soddisfa la proprietà possa non esistere nell'intervallo considerato richiede una complicazione della guardia e della postcondizione del ciclo. Vediamo un esempio.

Esempio 7.2 Dato un vettore di naturali a, determinare, se esiste, il minimo indice x nel dominio del vettore tale che $a[x] > 100$.

Osserviamo innanzitutto che l'intervallo di ricerca è il dominio dell'array a e che la proprietà in questione, $a[x] > 100$, è definita in $dom(a)$, ovvero

$$(\forall k \in dom(a).def(a[k] > 100)).$$

Stabiliamo innanzitutto che, nella postcondizione, la variabile x in questione assuma valore convenzionale n quando non esiste in a alcun elemento minore di 100. In altre parole specifichiamo il problema come segue (anche in questo caso a, n sono costanti).

$$\{\mathbf{tt}\}\ RLI\ \{x \in [0,n] \wedge (\forall j \in [0,x).\ a[j] \leq 100)\ \wedge\ (x < n \Rightarrow a[x] > 100)\}$$

Indichiamo ancora con *post* l'asserzione finale della specifica data. Nella costruzione dell'invariante e della guardia candidate per il ciclo che risolve il problema, procediamo come nel caso della ricerca lineare certa, individuando *Inv* e G tali che $Inv \wedge \neg G \Rightarrow post$. Un primo tentativo è il seguente:
Osserviamo che:

$$\begin{array}{ll}
 & x \in [0,n]\ \wedge\ (\forall j \in [0,x).\ a[j] \leq 100)\ \wedge\ (x < n \Rightarrow a[x] > 100) \\
\equiv & \quad \{\ \text{elim-}\Rightarrow\ \} \\
 & x \in [0,n]\ \wedge\ (\forall j \in [0,x).\ a[j] \leq 100)\ \wedge\ (x \geq n\ \vee\ a[x] > 100) \\
\equiv & \quad \{\ \text{De Morgan}\ \} \\
 & x \in [0,n]\ \wedge\ (\forall j \in [0,x).\ a[j] \leq 100)\ \wedge\ \neg(x < n\ \wedge\ a[x] \leq 100)
\end{array}$$

Possiamo allora scegliere:

$$Inv \equiv x \in [0,n]\ \wedge\ (\forall j \in [0,x).\ a[j] \leq 100)$$

$$G \equiv x < n \wedge a[x] \leq 100$$

La scelta della guardia presenta però un problema. Ricordiamo infatti che, nella determinazione dell'ipotesi di invarianza, dobbiamo assicurare che, dopo ogni iterazione del ciclo, la guardia sia definita. È chiaro invece che la precedente guardia non è definita quando $x = n$.

Per risolvere questo problema introduciamo una variabile booleana *trovato* e rimpiazziamo, nella guardia, $a[x] > 100$ con $\neg trovato$, assicurando, nell'invariante, che valga l'implicazione $trovato\ \Rightarrow\ a[x] > 100$. Otteniamo così le seguenti nuove candidate per l'invariante e la guardia:

$$Inv \equiv x \in [0,n]\ \wedge\ (\forall j \in [0,x).\ a[j] \leq 100)\ \wedge (trovato\ \Rightarrow\ a[x] > 100)$$

$$G \equiv x < n \wedge \neg trovato$$

Osserviamo infatti che:

$$\begin{array}{ll}
 & Inv\ \wedge\ \neg G \\
\equiv & \quad \{\ \text{definizione di } Inv \text{ e } G\ \} \\
 & x \in [0,n]\ \wedge\ (\forall j \in [0,x).\ a[j] \leq 100)\ \wedge\ (trovato\ \Rightarrow\ a[x] > 100)\ \wedge \neg(x < n \wedge \neg trovato) \\
\equiv & \quad \{\ \text{De Morgan}\ \} \\
 & x \in [0,n]\ \wedge\ (\forall j \in [0,x).\ a[j] \leq 100)\ \wedge\ (trovato\ \Rightarrow\ a[x] > 100)\ \wedge (x \geq n \vee trovato) \\
\equiv & \quad \{\ \text{elim-}\Rightarrow\ \} \\
 & x \in [0,n]\ \wedge\ (\forall j \in [0,x).\ a[j] \leq 100)\ \wedge\ (trovato\ \Rightarrow\ a[x] > 100)\ \wedge (x < n\ \Rightarrow\ trovato) \\
\Rightarrow & \quad \{\ \text{transitività di } \Rightarrow\ \} \\
 & x \in [0,n]\ \wedge\ (\forall j \in [0,x).\ a[j] \leq 100)\ \wedge\ (x < n\ \Rightarrow\ a[x] > 100) \\
\equiv & \quad \{\ \text{definizione di } post\ \} \\
 & post
\end{array}$$

Dunque le nuove candidate *Inv* e G non presentano il problema precedente poiché, in questo caso abbiamo $def((x < n \wedge \neg trovato)) \equiv \mathbf{tt}$.

Possiamo a questo punto stabilire che il programma *RLI* è del tipo

$\{\mathbf{tt}\}$
$\quad C';$
$\{\mathsf{inv} : Inv\}$ $\{\mathsf{ft} : ?\}$
$\quad$ **while** $x < n$ *and not trovato* **do**
$\qquad C$
$\quad$ **endw**
$\{post\}$

Osserviamo innanzitutto che un comando C' del tipo

$$x, trovato := 0, false$$

verifica la tripla

$\{\mathbf{tt}\}\ x, trovato := 0, false\ \{Inv\}.$

Per determinare il corpo del ciclo, C, facciamoci guidare dall'ipotesi di invarianza della regola ($while$) per il comando ripetitivo e distinguiamo due casi a seconda che $a[x]$ sia o meno maggiore di 100. Se $a[x] \leq 100$ allora:

$$\begin{array}{ll}
& Inv \ \wedge \ G \\
\equiv & \quad \{ \text{ definizione di } Inv \text{ e } G \ \} \\
& x \in [0,n] \ \wedge \ (\forall j \in [0,x).\ a[j] \leq 100) \ \wedge (trovato \ \Rightarrow \ a[x] > 100) \wedge x < n \wedge \neg trovato \\
\equiv & \quad \{ \ (x \in [0,n] \wedge x < n) \equiv x \in [0,n) \ \} \\
& x \in [0,n) \ \wedge \ (\forall j \in [0,x).\ a[j] \leq 100) \ \wedge (trovato \ \Rightarrow \ a[x] > 100) \ \wedge \ \neg trovato \\
\Rightarrow & \quad \{ \ \neg\, p \ \Rightarrow \ (p \ \Rightarrow \ q) \ \} \\
& x \in [0,n) \ \wedge \ (\forall j \in [0,x).\ a[j] \leq 100) \ \wedge \ \neg trovato \\
\equiv & \quad \{ \ \textbf{Ip: } a[x] \leq 100, \text{ Intervallo } \} \\
& x \in [0,n) \ \wedge \ (\forall j \in [0,x+1).\ a[j] \leq 100) \ \wedge \ \neg trovato \\
\equiv & \quad \{ \ x \in [0,n) \ \Rightarrow \ x+1 \in [0,n] \ \} \\
& x+1 \in [0,n] \ \wedge \ (\forall j \in [0,x+1).\ a[j] \leq 100) \ \wedge \ \neg trovato \\
\Rightarrow & \quad \{ \ \neg\, p \ \Rightarrow \ (p \ \Rightarrow \ q) \ \} \\
& x+1 \in [0,n] \ \wedge \ (\forall j \in [0,x+1).\ a[j] \leq 100) \ \wedge \ (trovato \ \Rightarrow \ a[x+1] > 100) \\
\equiv & \quad \{ \text{ Sostituzione } \} \\
& Inv_x^{x+1}
\end{array}$$

Dunque, nell'ipotesi $a[x] \leq 100$, il comando $x := x + 1$ mantiene l'invariante. Consideriamo ora il caso $a[x] > 100$.

$$\begin{array}{ll}
& Inv \ \wedge \ G \\
\equiv & \quad \{ \text{ definizione di } Inv \text{ e } G \ \} \\
& x \in [0,n] \ \wedge \ (\forall j \in [0,x).\ a[j] \leq 100) \ \wedge (trovato \ \Rightarrow \ a[x] > 100) \wedge x < n \wedge \neg trovato \\
\Rightarrow & \quad \{ \text{ sempl-}\wedge \ \} \\
& x \in [0,n] \ \wedge \ (\forall j \in [0,x).\ a[j] \leq 100) \ \wedge (trovato \ \Rightarrow \ a[x] > 100) \\
\Rightarrow & \quad \{ \ \textbf{Ip: } a[x] > 100 \ \} \\
& x \in [0,n) \ \wedge \ (\forall j \in [0,x).\ a[j] \leq 100) \ \wedge (\mathbf{tt} \ \Rightarrow \ a[x] > 100) \\
\equiv & \quad \{ \text{ Sostituzione } \} \\
& Inv_{trovato}^{true}
\end{array}$$

Dunque in questo secondo caso il comando *trovato* $:=$ *true* mantiene l'invariante. Il comando determinato fino a questo punto è dunque

$\{\mathbf{tt}\}$
$\quad C';$
$\{\mathsf{inv} : Inv\}$ $\{\mathsf{ft} : ?\}$
$\quad$ **while** $x < n$ *and not trovato* **do**

$\quad\quad\quad$ **if** $a[x] > 100$
$\quad\quad\quad\quad$ **then** *trovato* $:=$ *true*
$\quad\quad\quad\quad$ **else** $x := x + 1$
$\quad\quad\quad$ **fi**
$\quad\quad$ **endw**
$\quad$ $\{post\}$

Veniamo alla funzione di terminazione. Dobbiamo determinare una funzione t che verifichi i due fatti

(1) $\{Inv \wedge G \wedge t = V\}$ **if** ... **fi** $\{t < V\}$

(2) $Inv \wedge G \;\Rightarrow\; t > 0$

dove **if** ... **fi** sta per il corpo del ciclo individuato. Per la regola (if) del comando condizionale, (1) è conseguenza dei seguenti fatti:

(1.1) $\{Inv \wedge G \wedge t = V \wedge a[x] \leq 100\}$ $x := x + 1$ $\{t < V\}$

(1.2) $\{Inv \wedge G \wedge t = V \wedge a[x] > 100\}$ $trovato := true$ $\{t < V\}$

(1.3) $Inv \wedge G \wedge t = V \;\Rightarrow\; def(a[x] \leq 100)$

L'implicazione (1.3) è vera indipendentemente da t. Per soddisfare (1.1) e (1.2) occorre determinare una funzione t che descresca sia per effetto dell'incremento di x sia per l'effetto dell'assegnamento a *trovato*. L'incremento di x nel caso (1.1) suggerisce una funzione di terminazione che dipenda da $-x$ (cfr. ricerca lineare certa). L'assegnamento $trovato := true$ fa crescere la funzione $int(trovato)$ definita come segue:

$$int(y) = \begin{cases} 1 & \text{se } y = true \\ 0 & \text{se } y = false \end{cases}$$

osservando che G garantisce $trovato = false$ prima di ogni iterazione del ciclo. Dunque, per (1.2) t deve anche essere funzione di $-int(trovato)$. Poniamo, dunque

$$t = n - (x + int(trovato)).$$

Per (1.1) abbiamo:

$$\begin{array}{ll} & (n - (x + 1 + int(trovato))) < V \\ \Leftarrow & \quad \{ \text{ aritmetica } \} \\ & n - (x + int(trovato)) < V + 1 \\ \equiv & \quad \{ \textbf{ Ip: } n - (x + int(trovato)) = V \ \} \\ & V < V + 1 \end{array}$$

Per (1.2) abbiamo:

$$\begin{array}{ll} & (n - (x + int(true))) < V \\ \equiv & \quad \{ \text{ definizione di } int \ \} \\ & n - (x + 1) < V \\ \equiv & \quad \{ \text{ aritmetica } \} \\ & n - x < V + 1 \\ \equiv & \quad \{ \textbf{ Ip: } \neg trovato \text{ (ovvero } trovato = false\text{), definizione di } int \ \} \\ & n - (x + int(trovato)) < V + 1 \\ \equiv & \quad \{ \textbf{ Ip: } n - (x + int(trovato)) = V \ \} \\ & V < V + 1 \end{array}$$

Il procedimento sviluppato nell'esempio può essere facilmente generalizzato al caso di ricerca *incerta* del minimo elemento di un intervallo $[a, b)$ che soddisfa una data proprietà G (esprimibile come espressione booleana del linguaggio di programmazione).

Ricerca lineare incerta

$\{pre\}$
  $x, trovato := a, false;$
$\{\mathsf{inv} : Inv\}$ $\{\mathsf{ft} : b - (x + int(trovato))\}$
  while $x < b$ *and not* $trovato$ **do**
    if G
      then $trovato := true$
      else $x := x + 1$
    fi
  endw
$\{post\}$

dove

pre: $x \in [a, b) \;\Rightarrow\; def(G)$

post: $x \in [a, b] \;\wedge\; (\forall j \in [a, x).\ \neg G^{j}_{x}) \;\wedge (x < b \;\Rightarrow\; G)$

Inv: $x \in [a, b] \;\wedge\; (\forall j \in [a, x).\ \neg G^{j}_{x}) \;\wedge\; (trovato \;\Rightarrow\; G)$

e dove la funzione *int* è definita come segue:

$$int(y) = \begin{cases} 1 & \text{se } y = true \\ 0 & \text{se } y = false \end{cases}$$

7.3 Ricerca binaria

Anche in questo caso si tratta del problema di determinare il minimo elemento di un intervallo dato che soddisfa una proprietà. Consideriamo dapprima il caso della ricerca binaria certa a partire da un esempio.

Esempio 7.3 Dato un vettore *ordinato* di naturali distinti a, che contiene il valore 100, determinare l'indice x di tale elemento.

Il problema può essere specificato come segue, supponendo che l'array sia ordinato in senso crescente (nella specifica a, n sono costanti e I è una variabile di specifica).

$\{pre\}$ *BIN* $\{post\}$

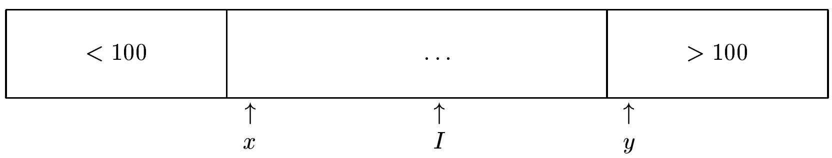
dove:

pre: $(\forall j \in [0, n-1).\ a[j] < a[j+1]) \wedge I \in [0, n) \wedge a[I] = 100$

post: $x = I$

Dal momento che l'array in questione deve rimanere costante e che, dunque, l'ipotesi di ordinamento è invariante per il problema, nel seguito ometteremo di esplicitare tale ipotesi nelle triple e la utilizzeremo, quando necessario, riferendola come $ord(a)$.

L'idea di base della ricerca binaria consiste nel mantenere un'invariante rappresentata dalla seguente figura



In altre parole, ad ogni iterazione, si garantisce che l'indice cercato, I, è nella porzione dell'array individuata dall'intervallo non vuoto $[x, y)$. Formalmente:

$$Inv : 0 \leq x < y \leq n \ \wedge \ I \in [x, y).$$

Osserviamo innanzitutto che un modo immediato per stabilire l'invariante all'inizio del ciclo consiste nell'assegnare i valori 0 e n rispettivamente a x e y, poiché l'ipotesi di certezza garantisce $I \in [0, n)$. Osserviamo inoltre:

$$
\begin{array}{ll}
 & 0 \leq x < y \leq n \ \wedge \ I \in [x, y) \wedge y = x + 1 \\
\Rightarrow & \quad \{ \text{ Leibniz, sempl-}\wedge \text{ } \} \\
 & I \in [x, x + 1) \\
\equiv & \quad \{ \text{ definizione di intervallo } \} \\
 & x = I
\end{array}
$$

Dunque scegliamo l'invariante e la guardia del ciclo nel modo seguente:

$$Inv \equiv 0 \leq x < y \leq n \ \wedge \ I \in [x, y)$$

$$G \equiv y \neq x + 1$$

e proviamo a sviluppare un comando del tipo

$\{pre\}$
  $x, y := 0, n;$
$\{\mathsf{inv} : Inv\}$ $\{\mathsf{ft} :?\}$
  while $y \neq x + 1$ **do**
    C
  endw
$\{post\}$

Per determinare C procediamo come segue. L'invariante e la guardia consentono di stabilire che l'intervallo $[x, y)$ contiene almeno due elementi (certamente x e $x + 1$) e che dunque esiste m con $x < m < y$. Osserviamo che:

$$
\begin{array}{ll}
 & I \in [x, y) \wedge x < m < y \\
\Rightarrow & \quad \{ \text{ Intervallo } \} \\
 & I \in [x, m) \vee I \in [m, y)
\end{array}
$$

Inoltre:

$$
\begin{array}{ll}
 & 0 \leq x < y \leq n \wedge x < m < y \\
\Rightarrow & \quad \{ \text{ Transitività } \} \\
 & 0 < m < n \\
\Rightarrow & \quad \{ \ def(a[m]) \equiv m \in [0, n) \ \} \\
 & def(a[m]).
\end{array}
$$

Utilizzando l'ipotesi $ord(a)$ e quest'ultima osservazione, a seconda che il valore di $a[m]$ sia o meno maggiore di 100, si possono stabilire i seguenti due fatti. Se $a[m] \leq 100$ allora:

$$
\begin{array}{ll}
 & I \in [x, m) \vee I \in [m, y) \\
\equiv & \quad \{ \ \textbf{Ip:} \ a[m] \leq 100 \ \wedge \ ord(a) \ \} \\
 & I \in [m, y) \\
\equiv & \quad \{ \text{ Sostituzione } \} \\
 & (I \in [x, y))^m_x
\end{array}
$$

Se invece $a[m] > 100$:

$$
\begin{array}{ll}
 & I \in [x,m) \vee I \in [m,y) \\
\equiv & \quad \{ \textbf{ Ip: } a[m] > 100 \ \wedge \ ord(a) \ \} \\
 & I \in [x,m) \\
\equiv & \quad \{ \text{ Sostituzione } \} \\
 & (I \in [x,y))^m_y
\end{array}
$$

Quanto detto sin qui non è altro che la dimostrazione di correttezza della tripla

$$\{Inv \wedge y \neq x+1 \wedge x < m < y\} \textbf{ if } a[m] \leq 100 \textbf{ then } x := m \textbf{ else } y := m \textbf{ fi } \{I \in [x,y)\}.$$

È facile vedere che il comando condizionale individuato soddisfa anche la tripla

$$
\begin{array}{l}
\{Inv \wedge y \neq x+1 \wedge x < m < y\} \\
\quad \textbf{if } a[m] \leq 100 \ \ \textbf{then } x := m \ \ \textbf{else } y := m \textbf{ fi} \\
\{0 \leq x < y \leq n\}
\end{array}
$$

e che, dunque, vale

$$\{Inv \wedge y \neq x+1 \wedge x < m < y\} \textbf{ if } a[m] \leq 100 \textbf{ then } x := m \textbf{ else } y := m \textbf{ fi } \{Inv\}$$

Possiamo quindi determinare, per il corpo C del ciclo, una sequenza del tipo

$$
\begin{array}{l}
\{Inv \wedge y \neq x+1\} \\
\qquad C'; \\
\qquad \textbf{if } a[m] \leq 100 \ \ \textbf{then } x := m \ \ \textbf{else } y := m \textbf{ fi} \\
\{Inv\}
\end{array}
$$

in modo che C' soddisfi la tripla

$$\{Inv \wedge y \neq x+1\} \ C' \ \{Inv \wedge y \neq x+1 \wedge x < m < y\}.$$

Il comando C' deve cioè assicurare che il comando condizionale venga intrapreso in uno stato in cui il valore di m sia un elemento di $[x,y)$ diverso da x. L'elemento di mezzo dell'intervallo $[x,y)$ soddisfa tale proprietà [4] e dunque C' può essere scelto come

$$m := (x+y) \ div \ 2.$$

Abbiamo in conclusione:

$$
\begin{array}{l}
\{Inv \wedge y \neq x+1\} \\
\quad m := (x+y) \ div \ 2; \\
\quad \textbf{if } a[m] \leq 100 \\
\qquad \textbf{then } x := m \\
\qquad \textbf{else } y := m \\
\quad \textbf{fi} \\
\{Inv\}
\end{array}
$$

Per quanto riguarda la terminazione del ciclo, osserviamo che l'ampiezza dell'intervallo $[x,y)$ si dimezza ad ogni iterazione. Scegliamo dunque, come funzione di terminazione t, la funzione $y-x$. Le ipotesi di terminazione e progresso della regola ($while$) sono allora:

(1) $Inv \wedge y \neq x+1 \Rightarrow y-x > 0$

(2) $\{Inv \wedge y \neq x+1 \wedge y-x = V\} \ C \ \{y-x < V\}$

La dimostrazione di (1) è ovvia; quella di (2) è lasciata per esercizio.

In conclusione il seguente programma risolve il problema specificato inizialmente, ovvero:

[4] Si ricordi che $y > x+1$, dunque l'intervallo contiene almeno due elementi.

$\{I \in [0, n) \wedge a[I] = 100 \wedge ord(a)\}$
$\quad x, y := 0, n;$
$\{\mathsf{inv} : Inv\}\ \{\mathsf{ft} : y - x\}$
$\quad$ **while** $y \neq x + 1$ **do**
$\qquad m := (x + y)\ div\ 2;$
$\qquad$ **if** $a[m] \leq 100$
$\qquad\quad$ **then** $x := m$
$\qquad\quad$ **else** $y := m$
$\qquad$ **fi**
$\quad$ **endw**
$\{0 \leq x < y \leq n \wedge I \in [x, x + 1)\}$
$\{x = I\}$

Il procedimento visto nell'esempio può essere esteso al caso di ricerca incerta. Consideriamo dapprima il caso della ricerca dell'elemento 100 nell'array *ordinato* a rimpiazzando l'ipotesi di certezza $(I \in [0, n) \wedge a[I] = 100)$ con l'ipotesi

$$a[0] \leq 100 < a[n - 1] \qquad\qquad (ip1).$$

Analogamente, rimpiazziamo l'asserto $I \in [x, y)$ dell'invariante visto in precedenza con l'asserto

$$a[x] \leq 100 < a[y]$$

osservando che anche in questo caso, grazie ad *(ip1)*, tale invariante può essere stabilito inizialmente con l'assegnamento

$x, y := 0, n - 1.$

Siano allora:

$Inv \equiv 0 \leq x < y \leq n - 1 \wedge a[x] \leq 100 < a[y]$

$post \equiv a[x] \leq 100 < a[x + 1]$

Si noti che, anche in questo caso, $Inv \wedge y = x + 1 \Rightarrow post$. È facile convincersi che Inv è ancora invariante per il ciclo definito in precedenza e che, dunque, il seguente programma è corretto:

$\{I \in [0, n) \wedge a[I] = 100\}$
$\quad x, y := 0, n - 1;$
$\{\mathsf{inv} : Inv\}\ \{\mathsf{ft} : y - x\}$
$\quad$ **while** $y \neq x + 1$ **do**
$\qquad m := (x + y)\ div\ 2;$
$\qquad$ **if** $a[m] \leq 100$
$\qquad\quad$ **then** $x := m$
$\qquad\quad$ **else** $y := m$
$\qquad$ **fi**
$\quad$ **endw**
$\{a[x] \leq 100 < a[x + 1]\}$

L'elemento cercato, se è presente in a, è dunque proprio $a[x]$.

Indeboliamo ora ulteriormente l'ipotesi *(ip1)*, rilasciando la condizione $a[n - 1] > 100$. In altre parole, consideriamo anche il caso in cui l'array a può contenere solo elementi minori o uguali a 100. Anche in questo caso, tuttavia, vorremmo poter sfruttare l'ipotesi di ordinamento dell'array per procedere, come prima, dimezzando ad ogni iterazione l'intervallo di ricerca. Non possiamo, però, mantenere nell'invariante una congiunzione del tipo $0 \leq x < y \leq n - 1 \wedge a[x] \leq 100\ \wedge 100 < a[y]$: non saremmo in grado, in generale, neanche di stabilirla inizialmente. Se però consentiamo che il valore di y (estremo destro dell'intervallo di ricerca) possa essere anche n, possiamo senz'altro stabilire inizialmente

$$0 \leq x < y \leq n \wedge a[x] \leq 100.$$

Si noti che $y = n$ non fa parte del dominio dell'array e dunque non possiamo rafforzare l'invariante con $a[y] > 100$. Possiamo, però, in analogia a qunato fatto per la ricerca lineare incerta, utilizzare la'sserto più debole, $y < n \;\Rightarrow\; a[y] > 100$. L'idea è che, non appena viene individuato un elemento dell'array che sia maggiore di 100, l'intervallo di ricerca, come nei casi precedenti, viene dimezzato modificandone l'estremo destro, che diviene quindi un indice nel dominio di a. In conclusione l'invariante diviene:

$$Inv \equiv 0 \leq x < y \leq n \wedge a[x] \leq 100 \wedge (y < n \Rightarrow 100 < a[y]).$$

Osserviamo ora che vale la seguente tripla:

$\{Inv \wedge y \neq x+1\}$
  $m := (x+y)\ div\ 2;$
  if $a[m] \leq 100$
    then $x := m$
    else $y := m$
  fi
$\{Inv\}$

Per dimostrare la tripla, procediamo in modo analogo a quanto fatto nel caso di ricerca certa. Osserviamo che vale la tripla:

$$\{Inv \wedge y \neq x+1\}\ m := (x+y)\ div\ 2\ \{Inv \wedge y \neq x+1 \wedge x < m < y\}.$$

Inoltre, dimostriamo la tripla

$$\{Inv \wedge y \neq x+1 \wedge m = (x+y)\ div\ 2\}\ \textbf{if} \ldots \textbf{fi}\ \{Inv\}.$$

utilizzando il teorema del condizionale e mostrando, cioè:

(1) $Inv \wedge y \neq x+1 \wedge m = (x+y)\ div\ 2 \Rightarrow def(a[m] \leq 100)$

(2) $\{Inv \wedge y \neq x+1 \wedge m = (x+y)\ div\ 2 \wedge a[m] \leq 100\}\ x := m\ \{Inv\}$

(3) $\{Inv \wedge y \neq x+1 \wedge m = (x+y)\ div\ 2 \wedge a[m] > 100\}\ y := m\ \{Inv\}$

Per quanto riguarda (1) abbiamo:

$$\begin{array}{ll} & 0 \leq x < y \leq n \wedge y \neq x+1 \wedge m = (x+y)\ div\ 2 \\ \Rightarrow & \quad \{ \text{ calcolo e transitività } \} \\ & 0 < m < n \\ \Rightarrow & \quad \{\ def(a[m] \leq 100) \equiv m \in [0,n)\ \} \\ & def(a[m] \leq 100) \end{array}$$

La dimostrazione di (2), analoga ai casi precedenti, è lasciata per esercizio. Per (3) e per la regola (ass) dell'assegnamento dobbiamo mostrare l'implicazione

$$\begin{array}{ll} & Inv \wedge y \neq x+1 \wedge m = (x+y)\ div\ 2 \wedge a[m] > 100 \\ \Rightarrow & \\ & 0 \leq x < m \leq n \wedge a[x] \leq 100 \wedge (m = n \Rightarrow a[m] > 100). \end{array}$$

Con dimostrazioni analoghe a quelle viste nei casi precedenti è facile vedere che vale l'implicazione:

$$\begin{array}{ll} & Inv \wedge y \neq x+1 \wedge m = (x+y)\ div\ 2 \wedge a[m] > 100 \\ \Rightarrow & \\ & 0 \leq x < m \leq n. \end{array}$$

e dunque basta mostrare

$$\begin{array}{ll} & Inv \wedge y \neq x+1 \wedge m = (x+y)\ div\ 2 \wedge a[m] > 100 \\ \Rightarrow & \end{array}$$

$a[x] \leq 100 \wedge (m = n \Rightarrow a[m] > 100)$,

che è banale.

Possiamo dunque concludere che il comando individuato in precedenza è corretto anche in questo caso. Si noti che:

$$\begin{array}{ll} & 0 \leq x < y \leq n \wedge a[x] \leq 100 \wedge (y < n \Rightarrow a[y] > 100) \wedge y = x + 1 \\ \Rightarrow & \quad \{ \text{ Leibniz, Sempl-}\wedge \text{ } \} \\ & x + 1 \leq n \wedge a[x] \leq 100 \wedge (x + 1 < n \Rightarrow a[x + 1] > 100). \end{array}$$

Anche in questo caso l'elemento cercato, se c'è, è proprio $a[x]$.

Generalizziamo ulteriormente il procedimento rilasciando anche l'ipotesi $a[0] \leq 100$: dunque l'array a, ordinato, può contenere solo elementi maggiori di 100. Dobbiamo rivedere di conseguenza l'invariante del ciclo precedente: non possiamo garantire, inizialmente $a[x] \leq 100$ con l'assegnamento $x := 0$. Consideriamo allora, in analogia a quanto fatto nel caso precedente, la seguente asserzione:

$$-1 \leq x < y \leq n \wedge (-1 < x \Rightarrow a[x] \leq 100) \wedge (y < n \Rightarrow 100 < a[y]),$$

che può essere stabilita inizialmente con l'assegnamento $x, y := -1, n$.

Osserviamo ora che vale ancora la tripla:

$\{Inv \wedge y \neq x + 1\}$
  $m := (x + y)\ div\ 2$;
  if $a[m] \leq 100$
    then $x := m$
    else $y := m$
  fi
$\{Inv\}$

Ragionando come nel caso precedente, la correttezza di tale tripla si basa sulla verifica dei seguenti tre fatti:

(1) $Inv \wedge y \neq x + 1 \wedge m = (x + y)\ div\ 2 \Rightarrow def(a[m] \leq 100)$

(2) $\{Inv \wedge y \neq x + 1 \wedge m = (x + y)\ div\ 2 \wedge a[m] \leq 100\}\ x := m\ \{Inv\}$

(3) $\{Inv \wedge y \neq x + 1 \wedge m = (x + y)\ div\ 2 \wedge a[m] > 100\}\ y := m\ \{Inv\}$

Per quanto riguarda (1) abbiamo:

$$\begin{array}{ll} & -1 \leq x < y \leq n \wedge y \neq x + 1 \wedge m = (x + y)\ div\ 2 \\ \Rightarrow & \quad \{ \text{ calcolo e transitività } \} \\ & -1 < m < n \\ \Rightarrow & \quad \{ \ def(a[m] \leq 100) \equiv m \in [0, n) \ \} \\ & def(a[m] \leq 100) \end{array}$$

Infine (2) e (3) sono semplici generalizzazioni delle dimostrazioni viste in precedenza e sono dunque lasciate per esercizio.

Analizziamo anche in questo caso l'asserzione finale rappresentata dal $Inv \wedge y = x + 1$. Abbiamo:

$$\begin{array}{ll} & -1 \leq x < y \leq n \wedge (-1 < x \Rightarrow a[x] \leq 100) \wedge (y < n \Rightarrow a[y] > 100) \wedge y = x + 1 \\ \equiv & \quad \{ \text{ Elim-}\Rightarrow \text{ } \} \\ & -1 \leq x < y \leq n \wedge (x \leq -1 \vee a[x] \leq 100) \wedge (y \geq n \vee a[y] > 100) \wedge y = x + 1 \\ \equiv & \quad \{ \ -1 \leq x \Rightarrow (x \leq -1 \equiv x = -1), y \leq n \Rightarrow (y \geq n \equiv y = n) \ \} \\ & -1 \leq x < y \leq n \wedge (x = -1 \vee a[x] \leq 100) \wedge (y = n \vee a[y] > 100) \wedge y = x + 1 \\ \Rightarrow & \quad \{ \text{ Leibniz, Sempl-}\wedge\text{, aritmetica } \} \\ & -1 \leq x < n \wedge (x = -1 \vee a[x] \leq 100) \wedge (x = n - 1 \vee a[x + 1] > 100). \end{array}$$

Manipoliamo ulteriormente la formula appena ottenuta considerando i seguenti tre casi: $x = -1$, $x = n-1$ e $x \in [0, n-1)$.

(i) $x = -1$.

$$
\begin{array}{ll}
 & -1 \leq x < n \wedge (x = -1 \vee a[x] \leq 100) \wedge (x = n-1 \vee a[x+1] > 100) \\
\Rightarrow & \quad \{ \textbf{Ip: } x = -1, \text{Leibniz, Zero, Unità, Sempl-}\wedge \} \\
 & x = -1 \wedge a[0] > 100 \\
\Rightarrow & \quad \{ \mathit{ord}(a) \} \\
 & x = -1 \wedge (\forall i \in [0, n). a[i] > 100)
\end{array}
$$

In questo caso, dunque, l'array a non contiene l'elemento 100.

(ii) $x = n - 1$.

$$
\begin{array}{ll}
 & -1 \leq x < n \wedge (x = -1 \vee a[x] \leq 100) \wedge (x = n-1 \vee a[x+1] > 100) \\
\Rightarrow & \quad \{ \textbf{Ip: } x = n-1, \text{Leibniz, Zero, Unità, Sempl-}\wedge \} \\
 & x = n-1 \wedge a[n-1] \leq 100 \\
\Rightarrow & \quad \{ \mathit{ord}(a) \} \\
 & x = n-1 \wedge (\forall i \in [0, n-1). a[i] < 100) \wedge a[n-1] \leq 100
\end{array}
$$

In questo caso, dunque, l'array a contiene solo elementi minori o uguali a 100 e l'elemento 100, se c'è, è proprio $a[n-1]$ ovvero $a[x]$.

(iii) $x \in [0, n-1)$.

$$
\begin{array}{ll}
 & -1 \leq x < n \wedge (x = -1 \vee a[x] \leq 100) \wedge (x = n-1 \vee a[x+1] > 100) \\
\Rightarrow & \quad \{ \textbf{Ip: } x \in [0, n-1), \text{Zero, Unità, Sempl-}\wedge \} \\
 & 0 \leq x < n-1 \wedge a[x] \leq 100 \wedge a[x+1] > 100 \\
\Rightarrow & \quad \{ \mathit{ord}(a) \} \\
 & x \in [0, n-1) \wedge (\forall i \in [0, x].\ a[i] \leq 100) \wedge (\forall i \in [x+1, n). a[i] > 100)
\end{array}
$$

In questo caso, dunque, l'array a contiene sia elementi minori o uguali a 100, sia elementi maggiori di 100 e l'elemento 100, se c'è, è proprio $a[x]$.

Si noti che quanto concluso nei casi (ii) e (iii) si può riassumere semplicemente con l'asserto:

$$x \in [0, n) \wedge (\forall i \in [0, x).\ a[i] < 100) \wedge a[x] \leq 100 \wedge (\forall i \in [x+1, n).\ a[i] > 100).$$

Concludiamo riassumendo quanto visto con il seguente frammento di programma di ricerca binaria di un elemento q in un array a ordinato in senso crescente.

Ricerca binaria di un elemento q in un array a ordinato

$$
\begin{array}{l}
\{\mathit{pre}\} \\
\quad x, y := -1, n; \\
\{\mathsf{inv} : \mathit{Inv}\}\{\mathsf{ft} : y - x\} \\
\quad \textbf{while } y \neq x + 1 \textbf{ do} \\
\quad\quad m := (x + y)\ \mathit{div}\ 2; \\
\quad\quad \textbf{if } a[m] \leq q \\
\quad\quad\quad \textbf{then } x := m \\
\quad\quad\quad \textbf{else } y := m \\
\quad\quad \textbf{fi} \\
\quad \textbf{endw} \\
\{\mathit{post}\}
\end{array}
$$

dove

pre: $(\forall j \in [0, n-1).\ a[j] < a[j+1])$

Inv: $-1 \leq x < y \leq n \wedge (x \geq 0 \Rightarrow a[x] \leq q) \wedge (y < n \Rightarrow a[y] > q))$.

post: $(x = -1 \wedge (\forall i \in [0, n).\ a[1] > q)) \vee$
$(x \in [0, n) \wedge (\forall i \in [0, x).\ a[i] \leq q) \wedge a[x] \leq q \wedge (\forall i \in [x+1, n).\ a[i] > q))$.

Il procedimento visto nel caso della ricerca di un elemento in un array ordinato può essere ulteriormente generalizzato. Consideriamo infatti una proprietà E *in funzione di* x ed esprimibile come espressione booleana del linguaggio di programmazione. Supponiamo innanzitutto che la condizione E sia definita nell'intervallo $[a, b)$. Supponiamo inoltre che tale proprietà verifichi i seguenti fatti:

(i) $x \in [a, b) \wedge E \Rightarrow (\forall i \in [a, x].E^i_x)$

(ii) $x \in [a, b) \wedge \neg E \Rightarrow (\forall i \in [x, b).\neg E^i_x)$.

Si noti che, nel caso di ricerca di un elemento q in un array ordinato, questi due fatti sono conseguenza dell'ipotesi di ordinamento, dove E non è altro che $a[x] \leq q$. Per analogia con l'esempio della ricerca in un array ordinato, indicheremo con $ord(a, E)$ l'ipotesi rappresentata dai due fatti precedenti.

La generalizzazione dell'esempio porta al seguente frammento di programma:

<u>Ricerca binaria</u>

$\{pre\}$
  $x, y := a-1, b$;
$\{\mathsf{inv} : Inv\}$ $\{\mathsf{ft} : y - x\}$
  while $y \neq x+1$ **do**
    $m := (x+y)\ div\ 2$;
    if E^m_x
      then $x := m$
      else $y := m$
    fi
  endw
$\{post\}$

dove

pre: $ord(a, E) \wedge (x \in [a, b) \Rightarrow def(E))$

Inv: $(a - 1 \leq x < y \leq b \wedge (x \geq a \Rightarrow E) \wedge (y < b \Rightarrow \neg E^y_x))$.

post: $(x = a - 1 \wedge (\forall i \in [a, b).\ \neg E^i_x)) \vee$
$(x \in [a, b) \wedge (\forall i \in [a, x).\ E^i_x) \wedge E \wedge (\forall i \in [x+1, b).\ \neg E^i_x))$.

La correttezza del frammento precedente si basa essenzialmente sui seguenti fatti:

(1) $Inv \wedge y \neq x+1 \wedge m = (x+y)\ div\ 2 \Rightarrow def(E^m_x)$

(2) $\{Inv \wedge y \neq x+1 \wedge m = (x+y)\ div\ 2 \wedge E^m_x\}\ x := m\ \{Inv\}$

(3) $\{Inv \wedge y \neq x+1 \wedge m = (x+y)\ div\ 2 \wedge \neg E^m_x\}\ y := m\ \{Inv\}$

Vediamo solo (2) lasciando (1) e (3) per esercizio. Per (ass) dobbiamo mostrare l'implicazione

$$Inv \wedge y \neq x+1 \wedge m = (x+y)\ div\ 2 \wedge E^m_x$$
$$\Rightarrow$$
$$a - 1 \leq m < y \leq b \wedge (m \geq a \Rightarrow E^m_x) \wedge (y < b \Rightarrow \neg E^y_x).$$

Osserviamo innanzitutto:

$$
\begin{array}{ll}
& a-1 \leq x < y \leq b \ \wedge \ y \neq x+1 \\
\Rightarrow & \quad \{ \text{ prop. } div \text{ e } Sempl\text{-}\wedge \ \} \\
& x < (x+y) \ div \ 2 < y \\
\Rightarrow & \quad \{ \ \mathbf{Ip:} \ m = (x+y) \ div \ 2, \text{ transitività } \} \\
& a-1 \leq x < m < y \leq b.
\end{array}
$$

Rimane dunque da dimostrare l'implicazione:

$$
\begin{array}{ll}
& Inv \wedge y \neq x+1 \wedge m = (x+y) \ div \ 2 \wedge E_x^m \\
\Rightarrow & \\
& (m \geq a \Rightarrow E_x^m) \wedge (y < b \Rightarrow \neg E_x^y).
\end{array}
$$

Abbiamo:

$$
\begin{array}{ll}
& Inv \wedge y \neq x+1 \wedge m = (x+y) \ div \ 2 \wedge E_x^m \\
\Rightarrow & \quad \{ \text{ definizione di } Inv, \text{ Sempl-}\wedge \ \} \\
& E_x^m \wedge (y < b \Rightarrow \neg E_x^y) \\
\Rightarrow & \quad \{ \text{ Intro-}\vee \ \} \\
& (m < a \vee E_x^m) \wedge (y < b \Rightarrow \neg E_x^y) \\
\equiv & \quad \{ \text{ Elim-}\Rightarrow \ \} \\
& (m \geq a \Rightarrow E_x^m) \wedge (y < b \Rightarrow \neg E_x^y).
\end{array}
$$

Nelle dimostrazioni precedenti non si è mai sfruttata l'ipotesi $ord(E)$ che, però, entra in gioco nel dimostrare che *post* è effettivamente una postcondizione per il ciclo dato. Con un ragionamento per casi analogo a quello effettuato nel caso della ricerca in un array ordinato, abbiamo infatti, nel caso $x \in [a, b-1)$:

$$
\begin{array}{ll}
& Inv \wedge y = x+1 \\
\equiv & \quad \{ \text{ Elim-}\Rightarrow \ \} \\
& a-1 \leq x < y \leq b \wedge (x \leq a-1 \vee E) \wedge (y \geq b \vee \neg E_x^y) \wedge y = x+1 \\
\equiv & \quad \{ \ a-1 \leq x \Rightarrow (x \leq a-1 \equiv x = a-1), y \leq b \Rightarrow (y \geq b \equiv y = b) \ \} \\
& a-1 \leq x < y \leq b \wedge (x = a-1 \vee E) \wedge (y = b \vee \neg E_x^y) \wedge y = x+1 \\
\Rightarrow & \quad \{ \text{ Leibniz, Sempl-}\wedge, \text{ aritmetica } \} \\
& a-1 \leq x < b \wedge (x = a-1 \vee E) \wedge (x = b-1 \vee \neg E_x^{x+1}) \\
\equiv & \quad \{ \ \mathbf{Ip:} \ x \in [a, b-1), \text{ Zero, Unità, Sempl-}\wedge \ \} \\
& a \leq x < b-1 \wedge E \wedge \neg E_x^{x+1} \\
\Rightarrow & \quad \{ \ ord(\text{E}) \ \} \\
& x \in [a, b-1) \wedge (\forall i \in [a, x+1). \ E_x^i) \wedge (\forall i \in [x+1, b). \ \neg E_x^i)
\end{array}
$$

Le dimostrazioni corrispondenti ai casi $x = a-1$ e $x = n-1$ sono lasciate per esercizio.

Considerazioni sull'efficienza

Si osservi che ogni problema che possa essere risolto applicando lo schema di ricerca binaria può essere anche risolto mediante una ricerca lineare. La ricerca binaria, però, presenta il vantaggio di essere più *efficiente*. Con riferimento allo schema generale, sia $N = b - a$ l'ampiezza del dominio della sequenza sulla quale si effettua la ricerca. Possiamo supporre, senza perdita di generalità, che N sia una potenza di due, ossia

$$N = 2^h,$$

e quindi inizialmente la funzione di terminazione ha valore $y - x = 2^h$. Inoltre è facile verificare che tale proprietà si mantiene invariante nel ciclo. L'osservazione banale, ma basilare, è che, se $d - c = 2^h$, con $h \geq 1$, allora

$$
\begin{array}{ll}
& (c+d) \ div \ 2 \\
= & \quad \{ \text{ calcolo } \} \\
& (d - c + 2 * c) \ div \ 2
\end{array}
$$

$$
\begin{array}{ll}
= & \quad \{\ \mathbf{Ip:}\ d - c = 2^h\ \} \\
& (2^h + 2 * c)\ div\ 2 \\
= & \quad \{\ \text{proprietà di } div\ \} \\
& 2^h\ div\ 2 + 2 * c\ div\ 2 \\
= & \quad \{\ \text{calcolo e } \mathbf{Ip:}\ h \geq 1\ \} \\
& 2^{h-1} + c.
\end{array}
$$

Questa osservazione permette di verificare facilmente la correttezza di:

$$
\begin{array}{l}
\{y - x = 2^K\ \wedge\ (K \geq 1)\} \\
\quad m := (x + y)\ div\ 2; \\
\{m = 2^{K-1} + x\} \\
\quad \mathbf{if}\ E_x^m \\
\quad\quad \mathbf{then}\ x := m \\
\quad\quad \mathbf{else}\ y := m \\
\quad \mathbf{fi} \\
\{y - x = 2^{K-1}\}
\end{array}
$$

e quindi di provare che $-1 \leq x < y \leq n\ \wedge\ \exists k.\ y - x = 2^k$ è invariante per il ciclo.

Da queste considerazioni segue che $\log_2(y - x)$ è una funzione intera dello stato, che decresce ad ogni iterazione e che è sempre ≥ 0. Quindi, $\log_2(y-x)$ è una funzione di terminazione e come tale il suo valore iniziale, cioè $\log_2(b - a)$, pone un limite superiore (logaritmico) al numero di iterazioni che, in generale, è molto più piccolo del limite $b - a$ relativo alla ricerca lineare. Quindi, quando sia possibile, ragioni di efficienza suggeriscono caldamente di applicare lo schema di ricerca binaria.

7.4 Esempi

Esempio 7.4 Dato un numero naturale m, calcolare la parte intera della sua radice quadrata. Il problema può essere specificato come segue:

$\{m \geq 0\}\ C\ \{x = \lfloor\sqrt{m}, \rfloor\}$ con m costante.

Osserviamo che:

$$
\begin{array}{ll}
& x = \lfloor\sqrt{m}\rfloor \\
\equiv & \quad \{\ \text{definizione di } \lfloor\sqrt{m}\rfloor\ \} \\
& x = min\{i \in [0, m+1) \mid m < (i+1)^2\}
\end{array}
$$

Dunque possiamo istanziare lo schema di *ricerca lineare certa* come segue:

$$
\begin{array}{l}
\{pre\} \\
\quad x := 0; \\
\{\mathsf{inv} : Inv\}\ \{\mathsf{ft} : I - x\} \\
\quad \mathbf{while}\ (x+1) * (x+1) \leq m\ \mathbf{do} \\
\quad\quad x := x + 1 \\
\quad \mathbf{endw} \\
\{post\} \\
\{x = \lfloor\sqrt{m}\rfloor\}
\end{array}
$$

dove

pre: $I \in [0, m+1) \wedge m < (I+1)^2$

post: $x \in [0, m+1)\ \wedge\ (\forall j \in [0, x).\ (j+1)^2 \leq m)\ \wedge\ (x+1)^2 > m$

Inv: $x \in [0, I)\ \wedge\ (\forall j \in [0, x).\ (j+1)^2 \leq m)$

Si osservi che in questo caso la sequenza alla quale si applica lo schema di ricerca è l'intervallo $[0, \lfloor\sqrt{m}\rfloor + 1)$ e non un vettore. O meglio, se si pensa astrattamente ad un vettore a come ad una funzione $a : dom(a) \to \mathbb{Z}$ allora l'intervallo in questione può essere visto come il vettore $a : [0, \lfloor\sqrt{m}\rfloor + 1) \to \mathbb{Z}$, tale che

$$a[i] = i, \quad \text{per ogni } i \in dom(a).$$

Si noti che questo è ovviamente un vettore ordinato rispetto alla proprietà di interesse, ovvero $a[i]^2 \leq m$, e quindi il problema può essere risolto anche applicando lo schema di ricerca binaria. Ecco il programma risultante:

$\{pre\}$
  $x, y := 0, m + 1$
$\{$inv :$Inv\}$ ft : y-x
  while $y \neq x + 1$ **do**
    $k := (x + y) \div 2;$
    if $k * k \leq m$
      then $x := m$
      else $y := m$
    fi
  endw
$\{post\}$

Laddove, la proprietà E è:

$$E \equiv x^2 \leq m,$$

e, indicando con a l'array che rappresenta l'intervallo $[0, \lfloor m + 1)$, definito come sopra ($a[i] = i$ per ogni $i \in [0, m + 1)$:

pre: $ord(a, E) \ \wedge \ (x \in [0, m + 1) \ \Rightarrow \ def(x^2 \leq m))$

Inv: $0 \leq x < y < m + 1 \ \wedge \ x^2 \leq m \ \wedge y^2 > m$

$post$: $x \in [0, m + 1) \ \wedge \ \forall i \in [0, x).\ i^2 \leq m \ \wedge \forall i \in x + 1, m + 1).i^2 > m$

Esempio 7.5 Determinare un comando C che soddisfi la seguente tripla:

$\{n > 0\}\ C\ \{b \equiv (\forall i \in [1, n).\ pari(a[i]))\}$,

dove a : **array** $[0, n)$ **of int** e a ed n sono costanti.

Osserviamo che:

$$\begin{array}{ll} & (\forall i \in [1, n).\ pari(a[i])) \\ \equiv & \quad \{ \text{ De Morgan } \} \\ & \neg(\exists i \in [1, n).\ dispari(a[i])) \end{array}$$

Possiamo dunque utilizzare lo schema di ricerca lineare incerta per determinare, se esiste, il minimo elemento in $[1, n)$ tale che $dispari(a[i])$. In altre parole, indicando con RLI l'istanza dello schema in questo caso, sviluppiamo un comando del tipo:

$\{n > 0\}$
  RLI;
$\{post\}$
  $b := (x = n)$
$\{b \equiv (\forall i \in [1, n).\ pari(a[i]))\}$

dove

post: $x \in [1, n] \ \wedge \ (\forall j \in [1, x).\ pari(a[x])) \ \wedge (x < n \ \Rightarrow \ dispari(a[x]))$.

La correttezza di questo frammento di programma si riduce semplicemente alla dimostrazione della tripla:

$\{post\}\ b := (x = n)\ \{b \equiv (\forall i \in [1, n).\ pari(a[i]))\}$.

ovvero (ricordando la regola (ass) e la definizione di *post*) dell'implicazione:

$$
\begin{array}{l}
\quad x \in [1, n] \ \wedge \ (\forall j \in [1, x).\ pari(a[x])) \ \wedge (x < n \ \Rightarrow \ dispari(a[x])) \\
\Rightarrow \\
\quad (x = n) \equiv (\forall i \in [1, n).\ pari(a[i]))
\end{array}
$$

Tale implicazione può essere dimostrata, per casi, come segue:

(i) se $x = n$ allora:

$$
\begin{array}{ll}
 & x \in [1, n] \ \wedge \ (\forall j \in [1, x).\ pari(a[j])) \ \wedge (x < n \ \Rightarrow \ dispari(a[x])) \\
\Rightarrow & \qquad \{ \text{ Sempl-}\wedge \text{ } \} \\
 & x \in [1, n] \ \wedge \ (\forall j \in [1, x).\ pari(a[j])) \\
\Rightarrow & \qquad \{ \textbf{ Ip: } x = n, \text{ Leibniz } \} \\
 & (\forall j \in [1, n).\ pari(a[j])) \\
\equiv & \qquad \{ \ p \equiv (\mathbf{tt} \equiv p) \ \} \\
 & \mathbf{tt} \equiv (\forall j \in [1, n).\ pari(a[j])) \\
\equiv & \qquad \{ \textbf{ Ip: } x = n \ \} \\
 & (x = n) \equiv (\forall j \in [1, n).\ pari(a[j]))
\end{array}
$$

(ii) se $x \neq n$ allora:

$$
\begin{array}{ll}
 & x \in [1, n] \ \wedge \ (\forall j \in [1, x).\ pari(a[x])) \ \wedge (x < n \ \Rightarrow \ dispari(a[x])) \\
\Rightarrow & \qquad \{ \textbf{ Ip: } x \neq n, \text{ Sempl-}\wedge \text{ } \} \\
 & x \in [1, n) \wedge dispari(a[x]) \\
\Rightarrow & \qquad \{ \ P_i^x \Rightarrow (\exists i.P) \ \} \\
 & (\exists i.\ i \in [1, n) \wedge dispari(a[i])) \\
\equiv & \qquad \{ \text{ De Morgan } \} \\
 & \neg(\forall i \in [1, n).\ pari(a[i])) \\
\equiv & \qquad \{ \ \neg p \equiv (\mathbf{ff} \equiv p) \ \} \\
 & \mathbf{ff} \equiv (\forall i \in [1, n).\ pari(a[i])) \\
\equiv & \qquad \{ \textbf{ Ip: } x \neq n \ \} \\
 & (x = n) \equiv (\forall i \in [1, n).\ pari(a[i]))
\end{array}
$$

Riportiamo di seguito l'istanza di *RLI* utilizzata nella soluzione del problema

```
x, trovato := 0, false;
while x < n and not trovato do
    if a[x] mod 2 ≠ 0
      then trovato := true
      else x := x + 1
    fi
endw
```

Esempio 7.6 È dato un array di interi distinti con dominio $[0, n)$, ordinato in modo *crescente*. Si vuole determinare il minimo indice, se esiste, minore o uguale dell'elemento corrispondente.

Supponiamo che l'array contenga almeno due elementi, altrimenti il problema si risolve semplicemente con un comando condizionale. La precondizione del problema può essere espressa nel seguente modo (assumendo a : **array** $[0, n)$ **of int**).

$\{(\forall i \in [0, n-1).\ a[i] \ < a[i+1])\}$.

Indichiamo con $ord(a)$ tale proprietà. Come nel caso della ricerca lineare incerta specifichiamo la postcondizione come:

$\{post : x \in [0,n] \wedge (x < n \Rightarrow (x = min\{i \in [0,n) \mid i \leq a[i]\}))\}$.

Anziché utilizzare la ricerca lineare incerta, osserviamo che con un semplice ragionamento induttivo è facile vedere che valgono i seguenti due fatti:

(1) $x \in [0,n) \wedge x > a[x]$ $\Rightarrow$ $(\forall i \in [0,x].\ x > a[x])$	(2) $x \in [0,n) \wedge x \leq a[x]$ $\Rightarrow$ $(\forall i \in [x,n).\ x \leq a[x])$

Proviamo, ad esempio, la proprietà (1). Sia $x \in [0,n)$ con $x > a[x]$ e anche $x > 0$ (per $x = 0$ la proprietà è vera banalmente). Abbiamo allora:

$$
\begin{array}{ll}
 & x > a[x] \\
\equiv & \quad \{ \text{ calcolo } \} \\
 & x - 1 \geq a[x] \\
\Rightarrow & \quad \{ \textbf{ Ip: } ord(a) \} \\
 & x - 1 \geq a[x] > a[x-1] \\
\Rightarrow & \quad \{ \text{ transitività } \} \\
 & x - 1 > a[x-1].
\end{array}
$$

Con una semplice induzione possiamo concludere che vale (1). Per (2) il ragionamento è del tutto simile.

Le proprietà (1) e (2) consentono di applicare una ricerca binaria per la soluzione del problema, utilizzando la condizione $a[x] < x$. Otteniamo così il seguente frammento di programma:

```
{pre}
   x, y := −1, n;
{inv : Inv} {ft : y − x}
   while y ≠ x + 1 do
      m := (x + y) div 2;
      if a[m] < m
         then x := m
         else y := m
      fi
   endw
{post₁}
```

dove

pre: $ord(a)$

Inv: $(-1 \leq x < y \leq n \wedge (x \geq 0 \Rightarrow a[x] < x) \wedge (y < n \Rightarrow a[x] \geq x))$.

$post_1$: $(x = -1 \wedge (\forall i \in [0,n).\ a[i] \geq i) \vee$
$(x \in [0,n) \wedge (\forall i \in [0,x].\ i > a[i]) \wedge (\forall i \in [x+1,b).\ i \leq a[i]))$.

Dunque, abbiamo che l'indice cercato esiste se $x + 1 \neq n$ ed è proprio $x + 1$. Possiamo dunque far seguire alla ricerca binaria il comando di assegnamento $x := x + 1$ che soddisfa chiaramente:

$\{post_1\}\ x := x + 1\ \{post\}$

8 Tecniche di programmazione iterativa

Il problema principale nella dimostrazione di correttezza di programmi che coinvolgono cicli e nella determinazione dei cicli stessi, consiste quasi sempre nell'individuare l'invariante *giusta*. Pur non potendo definire un modo sistematico per derivare l'invariante, possiamo suggerire alcune tecniche che risultano di grande utilità nella pratica programmativa. Queste tecniche si basano essenzialmente sull'analisi delle premesse della regola ($while$). In particolare, data una specifica del tipo

$\{pre\}\ C\ \{post\}$

si tenta di analizzare *post* in modo da individuare una congiunzione del tipo $P \wedge G$, tale che:

- $P \wedge G \Rightarrow post$
- P possa essere scelta come invariante candidata di un ciclo
- G possa essere espressa come $\neg E$, dove E è una espressione booleana del linguaggio

Quindi, la regola ($while$) permette di concludere che un ciclo che mantenga P come invariante, che termina e la cui guardia sia E, porta in uno stato che soddisfa la *post*.

Vediamo subito un esempio:

Esempio 8.1 È data la seguente specifica

$\{x \geq 0 \wedge y > 0\}\ C\ \{q = x\ div\ y\}$ con x, y costanti.

Osserviamo che:

$$\begin{array}{ll} & q = x\ div\ y \\ \equiv & \quad \{ \text{ definizione di } div \ \} \\ & (\exists r.\ x = q \cdot y + r \wedge 0 \leq r < y) \end{array}$$

Introduciamo allora una nuova *variabile di programma* r che rappresenta la variabile quantificata dell'asserto precedente e riscriviamo la *post* come

$post : x = q \cdot y + r \wedge 0 \leq r \wedge r < y.$

Poniamo:

Inv: $x = q \cdot y + r \wedge 0 \leq r$

G: $r < y$.

Si tratta allora di risolvere le incognite del seguente programma annotato, ricordando che x, y non devono subire modifiche.

$$\begin{array}{l} \{x \geq 0 \wedge y > 0\} \\ \quad C_0; \\ \{\mathsf{inv} : x = q \cdot y + r \wedge 0 \leq r\}\ \{\mathsf{ft} :?\} \\ \quad \textbf{while } r \geq y \textbf{ do} \\ \qquad \{x = q \cdot y + r \wedge 0 \leq r \wedge r \geq y\} \\ \qquad C \\ \quad \textbf{endw} \\ \{x = q \cdot y + r \wedge 0 \leq r < y\} \\ \{q = x\ div\ y\} \end{array}$$

La prima osservazione riguarda il fatto che C_0 deve portare in uno stato in cui vale l'invariante. Vale chiaramente:

$\{x \geq 0 \wedge y > 0\}\ q, r := 0, x\ \{Inv\}.$

Ragioniamo ora sulla funzione di terminazione: in tutti gli stati in cui vale la guardia del ciclo, $r \geq y$, essendo y costante e $y > 0$, vale chiaramente $r > 0$. Se scegliamo proprio r come funzione di terminazione, dobbiamo fare in modo che C faccia diminuire r. Ma C deve anche mantenere *Inv* invariante, ovvero deve soddisfare la tripla

$$\{Inv \wedge r \geq y\}\ C\ \{Inv\}.$$

Osserviamo che:

$$
\begin{array}{ll}
 & x = q \cdot y + r \wedge r \geq y \\
\Rightarrow & \quad \{ \text{ calcolo } \} \\
 & x = (q+1) \cdot y + r - y \wedge r - y \geq 0 \\
\equiv & \quad \{ \text{ Sostituzione } \} \\
 & (x = q \cdot y + r \wedge r \geq 0)^{q+1, r-y}_{q,r}.
\end{array}
$$

Dunque il comando C cercato può essere $q, r := q+1, r-y$. Basta solo verificare che la funzione di terminazione decresce a seguito dell'esecuzione di C, ovvero che vale la tripla

$$\{Inv \wedge r \geq y \wedge r = V\}\ q, r := q+1, r-y\ \{r < V\}.$$

Ma:

$$
\begin{array}{ll}
 & (r < V)^{r-y}_{r} \\
\equiv & \quad \{ \text{ Sostituzione } \} \\
 & r - y < V \\
\equiv & \quad \{ \textbf{ Ip: } r = V \} \\
 & V - y < V \\
\equiv & \quad \{ y > 0 \} \\
 & \mathbf{tt}.
\end{array}
$$

Complessivamente, abbiamo dunque il programma:

$$
\begin{array}{l}
\{x \geq 0 \wedge y > 0\} \\
\quad q, r := 0, x; \\
\{\mathsf{inv} : x = q \cdot y + r \wedge 0 \leq r\}\ \{\mathsf{ft} : r\} \\
\quad \textbf{while } r \geq y \textbf{ do} \\
\qquad \{x = q \cdot y + r \wedge 0 \leq r \wedge r \geq y\} \\
\qquad q, r := q+1, r-y \\
\quad \textbf{endw} \\
\{x = q \cdot y + r \wedge 0 \leq r < y\} \\
\{q = x\ div\ y\}
\end{array}
$$

Nell'esempio visto è stato sufficiente ragionare in modo semplice sulla postcondizione per determinare l'invariante e la guardia candidate. Nel seguito mostreremo alcuni esempi di altre tecniche che si rivelano utili nella pratica per lo stesso scopo.

Rimpiazzamento di una costante con una variabile

Esempio 8.2 Consideriamo il problema di calcolare la somma degli elementi di un array di interi. Il problema può essere specificato come segue.

$\{n \geq 0\}\ C\ \{s = (\sum i : i \in [0, n).a[i])\}$ con n costante e a : **array** $[0, n)$ **of int**.

Rimpiazziamo nella postcondizione la costante n con una variabile x e osserviamo che

$$
\begin{array}{ll}
 & s = (\sum i : i \in [0, x).a[i]) \wedge x = n \\
\Rightarrow & \quad \{ \text{ Leibniz } \} \\
 & s = (\sum i : i \in [0, n).a[i])
\end{array}
$$

Scegliamo dunque come invariante e guardia candidate del ciclo

(i) $s = (\sum i : i \in [0, x).a[i])$

(ii) $x \neq n$

Si noti che un modo per stabilire inizialmente l'asserzione (i) consiste nell'assegnare alle variabili s e x il valore 0 (essendo 0 l'elemento neutro per la somma). Dunque la variabile x introdotta assume valore 0 all'inizio e valore n alla fine del ciclo. Quest'ultimo deve quindi far sì che il valore di x aumenti ad ogni iterazione, avvicinandosi al valore n. Ciò significa che una potenziale funzione di terminazione per il ciclo che stiamo sviluppando è $n-x$. Osserviamo tuttavia che, con l'invariante e la guardia scelte, non siamo in grado di dimostrare che tale funzione assume valori non negativi negli stati in cui vale l'invariante stessa. Possiamo tuttavia rafforzare quest'ultima imponendo che i valori di x siano compresi nell'intervallo $[0, n]$. Abbiamo cioè un ciclo parzialmente sviluppato del tipo:

$$
\begin{array}{l}
\{n \geq 0\} \\
\quad s, x := 0, 0; \\
\{\mathsf{inv} : s = (\sum i : i \in [0, x).a[i]) \wedge x \in [0, n]\} \ \{\mathsf{ft} : n - x\} \\
\quad \textbf{while } x \neq n \textbf{ do} \\
\qquad \{s = (\sum i : i \in [0, x).a[i]) \wedge x \in [0, n)\} \\
\qquad C \\
\quad \textbf{endw} \\
\{s = (\sum i : i \in [0, x).a[i]) \wedge x = n\} \\
\{s = (\sum i : i \in [0, n).a[i])\}
\end{array}
$$

Per determinare C osserviamo che esso deve far crescere il valore di x, ad esempio aumentandone di 1 il valore. Osserviamo allora che

$$
\begin{array}{ll}
 & s = (\sum i : i \in [0, x).\ a[i]) \\
\Rightarrow & \quad \{\ \textbf{Ip: } x \in [0, n), \text{ intervallo }\} \\
 & s + a[x] = (\sum i : i \in [0, x+1).\ a[i])
\end{array}
$$

Dunque, sommando ad s il valore $a[x]$, l'invariante viene mantenuta. Si osservi che l'ipotesi $x \in [0, n)$, che corrisponde a $\mathit{def}(a[x])$ è garantita dall'invariante e dalla guardia (si veda il programma annotato precedente). Abbiamo complessivamente:

$$
\begin{array}{l}
\{n \geq 0\} \\
\quad s, x := 0, 0; \\
\{\mathsf{inv} : s = (\sum i : i \in [0, x).a[i]) \wedge x \in [0, n]\} \ \{\mathsf{ft} : n - x\} \\
\quad \textbf{while } x \neq n \textbf{ do} \\
\qquad \{s = (\sum i : i \in [0, x).a[i]) \wedge x \in [0, n)\} \\
\qquad s, x := s + a[x], x + 1 \\
\quad \textbf{endw} \\
\{s = (\sum i : i \in [0, x).a[i]) \wedge x = n\} \\
\{s = (\sum i : i \in [0, n).a[i])\}
\end{array}
$$

Esempio 8.3 Vediamo un altro esempio di applicazione di tale tecnica. Sia data la tripla:

$$\{a \geq 0 \wedge b \geq 0\}\ C\ \{r = a^b\},$$

con a, b costanti.

Rimpiazziamo b (costante del problema) con una nuova variabile di programma x e dunque scegliamo come invariante e guardia candidate

- $r = a^x$
- $x \neq b$.

Anche in questo caso l'invariante candidata può essere facilmente stabilita all'inizio con l'assegnamento $r, x := 1, 0$. Il programma candidato a risolvere il problema è dunque:

$$
\begin{array}{l}
\{a \geq 0 \wedge b \geq 0\} \\
\quad r, x := 1, 0; \\
\{\mathsf{inv} : r = a^x\} \ \{\mathsf{ft} :?\} \\
\quad \textbf{while } x \neq b \textbf{ do} \\
\qquad \{r = a^x \wedge x \neq b\} \\
\qquad C \\
\quad \textbf{endw} \\
\{r = a^x \wedge x = b\} \\
\{r = a^b\}
\end{array}
$$

Ragioniamo sulla tripla

$$\{r = a^x \wedge x \neq b\}\ C\ \{r = a^x\}$$

che deve essere soddisfatta da C. Osserviamo che:

$$
\begin{array}{ll}
 & r = a^x \\
\equiv & \quad \{ \text{ calcolo } \} \\
 & r \cdot a = a^{x+1} \\
\equiv & \quad \{ \text{ sostituzione } \} \\
 & (r = a^x)^{r*a,x+1}_{r,x}
\end{array}
$$

Dunque il comando $r, x := r * a, x + 1$ mantiene l'invariante scelta. Per quanto riguarda la terminazione, osserviamo che x "si avvicina" a b e dunque $b - x$ sembra essere una buona candidata. È facile però vedere che non abbiamo modo di dimostrare che tale funzione è limitata inferiormente (ovvero che $Inv \Rightarrow b - x \geq 0$) a meno di rafforzare l'invariante. Osservando che, essendo b una costante non negativa, i valori che x assume durante l'esecuzione del ciclo stanno nell'intervallo $[0, b]$ e dunque possiamo considerare la seguente asserzione invariante:

$$Inv : r = a^x \wedge x \in [0, b].$$

In questo modo è evidente che $b - x$ è una buona funzione di terminazione, mentre va dimostrato che il comando preserva la nuova invariante. Basta però dimostrare che vale la tripla

$$\{r = a^x \wedge x \in [0, b] \wedge x \neq b\}\ r, x := r * a, x + 1\ \{x \in [0, b]\}.$$

Ma:

$$
\begin{array}{ll}
 & x \in [0, b] \wedge x \neq b \\
\equiv & \quad \{ \text{ intervallo } \} \\
 & x \in [0, b) \\
\Rightarrow & \quad \{ \text{ calcolo } \} \\
 & x + 1 \in [0, b]
\end{array}
$$

Rafforzamento di invarianti

Succede spesso che le tecniche viste in precedenza, o più in generale il tentativo di individuare una congiunzione del tipo $Inv \wedge \neg E$ che implichi l'asserzione finale di un problema dato, portino a situazioni problematiche nel determinare un comando che mantenga l'invariante candidata.

Abbiamo già visto due esempi nei quali l'invariante scelta inizialmente è stata rafforzata al fine di limitare i valori di una variabile introdotta per rimpiazzare una costante del problema. In altri casi però può succedere che:

- la guardia candidata non è direttamente esprimibile nel linguaggio di programmazione
- il comando desiderato come corpo del ciclo richiede il calcolo di espressioni non direttamente esprimibili nel linguaggio di programmazione

Di solito, la soluzione a questi problemi si ottiene introducendo una o più variabili nuove e rafforzando l'invariante con uguaglianze del tipo $x = E$, dove x è una variabile nuova introdotta ed E è un'espressione non direttamente esprimibile nel linguaggio. Chiaramente, il corpo del ciclo dovrà ora tener conto anche di queste porzioni dell'invariante.

Seguono alcuni esempi che illustrano tale situazione.

Esempio 8.4 Dato un array di interi a, si vuole determinare, se esiste, un indice x nell'array il cui elemento corrispondente è maggiore della somma di tutti gli elementi che lo precedono. Si tratta chiaramente di un problema di ricerca lineare incerta che può essere formulato come segue:

$$\{n \geq 0\}\ C\ \{x \in [0,n] \wedge (x < n \Rightarrow a[x] > \sum_{i=0}^{x-1} a[i])\},$$

con n costante e a : **array** $[0,n)$ **of int**.

Se l'espressione $a[x] > \sum_{i=0}^{x-1} a[i]$ fosse esprimibile nel linguaggio di programmazione, potremmo applicare direttamente lo schema di ricerca lineare incerta ottenendo il programma:

$\{pre\}$
    $x, trovato := 0, false;$
$\{\mathsf{inv} : Inv\}\ \{\mathsf{ft} : n - (x + int(trovato))\}$
    while $x < n$ *and not* $trovato$ **do**
        if $a[x] > \sum_{i=0}^{x-1} a[i]$
            then $trovato := true$
            else $x := x + 1$
        fi
    endw
$\{post\}$

dove

pre: $n \geq 0$

post: $x \in [0,n]\ \wedge\ (\forall j \in [0,x).\ a[j] \leq \sum_{i=0}^{j-1} a[i])\ \wedge (x < n \Rightarrow a[x] > \sum_{i=0}^{x-1} a[i])$

Inv: $x \in [0,n]\ \wedge\ (\forall j \in [0,x).\ a[j] \leq \sum_{i=0}^{j-1} a[i])\ \wedge\ (trovato \Rightarrow a[x] > \sum_{i=0}^{x-1} a[i])$

Introduciamo dunque una nuova variabile di programma s e rafforziamo l'invariante con l'uguaglianza:

$$s = \sum_{i=0}^{x-1} a[i],$$

dove per $x = 0$ la sommatoria è convenzionalmente pari a 0. Possiamo allora rimpiazzare la guardia del comando condizionale precedente con $a[x] > s$ che è ora un'espressione lecita del linguaggio. Per mantenere quest'invariante modificata, dobbiamo però rimpiazzare il comando $x := x + 1$ nel ramo **else** del comando condizionale con un nuovo comando che mantenga invariante anche l'uguaglianza $s = \sum_{i=0}^{x-1} a[i]$. È facile convincersi che tale comando può semplicemente essere l'assegnamento multiplo $x, s := x + 1, s + a[x]$ (verificarlo per esercizio). Analogamente, il comando di inizializzazione deve tener conto anche dell'inizializzazione di s, data da $s := 0$. Otteniamo così:

$$
\begin{array}{l}
\{pre\} \\
\quad s, x, trovato := 0, 0, false; \\
\{\mathsf{inv} : Inv\}\ \{\mathsf{ft} : n - (x + int(trovato))\} \\
\quad \textbf{while } x < n \textit{ and not trovato } \textbf{do} \\
\qquad \textbf{if } a[x] > s \\
\qquad\quad \textbf{then } trovato := true \\
\qquad\quad \textbf{else } x, s := x + 1, s + a[x] \\
\qquad \textbf{fi} \\
\quad \textbf{endw} \\
\{post\}
\end{array}
$$

dove

pre: $n \geq 0$

post: $x \in [0, n] \ \wedge\ (\forall j \in [0, x).\ a[j] \leq \sum_{i=0}^{j-1} a[i])\ \wedge (x < n\ \Rightarrow\ a[x] > \sum_{i=0}^{x-1} a[i])$

Inv: $x \in [0, n] \ \wedge\ (\forall j \in [0, x).\ a[j] \leq \sum_{i=0}^{j-1} a[i])\ \wedge s = \sum_{i=0}^{x-1} a[i]\ \wedge\ (trovato\ \Rightarrow\ a[x] > \sum_{i=0}^{x-1} a[i])$

Esempio 8.5 Consideriamo il problema di calcolare il termine k-esimo della successione di Fibonacci, con k costante naturale non nulla. Come è noto la successione di Fibonacci può essere espressa come segue:

$$
fib(n) = \begin{cases} 0, & \text{se } n = 0; \\ 1, & \text{se } n = 1; \\ fib(n-1) + fib(n-2), & \text{se } n \geq 2. \end{cases}
$$

Il problema può essere allora espresso come segue:

$$\{k \geq 1\}\ C\ \{x = fib(k)\},$$

con k costante.

Analogamente agli esempi precedenti possiamo scegliere come candidati per invariante e guardia:

Inv: $x = fib(i) \wedge i \in [1, k]$

E: $i \neq k$

e sviluppare un ciclo del tipo:

$$
\begin{array}{l}
\{k \geq 1\} \\
\quad C_0; \\
\{\mathsf{inv} : x = fib(i) \wedge i \in [1, k]\}\ \{\mathsf{ft} : k - i\} \\
\quad \textbf{while } i \neq k \textbf{ do} \\
\qquad \{x = fib(i) \wedge i \in [1, k)\} \\
\qquad C \\
\quad \textbf{endw} \\
\{x = fib(i) \wedge i = k\} \\
\{x = fib(k)\}
\end{array}
$$

Osserviamo innanzitutto che l'invariante scelta può essere inizializzata facilmente ponendo per C_0 l'assegnamento $x, i := 1, 1$. La funzione di terminazione scelta decresce al crescere di i e dunque possiamo costruire C come una sequenza del tipo $C'; i := i + 1$. Applicando la regola (*seq*) osserviamo che allora C' deve soddisfare la tripla

$$\{x = fib(i)\}\ C'\ \{x = fib(i + 1)\}.$$

Ora, osservando che:

$$
\begin{array}{ll}
 & \mathit{fib}(i+1) \\
= & \quad \{\ \textbf{Ip:}\ i \geq 1, \text{ definizione di } \mathit{fib}\ \} \\
 & \mathit{fib}(i) + \mathit{fib}(i-1) \\
= & \quad \{\ \textbf{Ip:}\ x = \mathit{fib}(i)\ \} \\
 & x + \mathit{fib}(i-1)
\end{array}
$$

un comando C' candidato potrebbe essere $x := x + \mathit{fib}(i-1)$, in cui però compare un'espressione, $\mathit{fib}(i-1)$ non direttamente esprimibile nel linguaggio. Introduciamo allora una nuova variabile di programma, y, e rafforziamo l'invariante con l'uguaglianza $y = \mathit{fib}(i-1)$. Il comando C' deve allora soddisfare la tripla:

$$\{x = \mathit{fib}(i) \wedge y = \mathit{fib}(i-1)\}\ C'\ \{x = \mathit{fib}(i+1) \wedge y = \mathit{fib}(i)\}.$$

Ma:

$$
\begin{array}{ll}
 & x = \mathit{fib}(i) \wedge y = \mathit{fib}(i-1) \\
\Rightarrow & \quad \{\ \textbf{Ip:}\ i \geq 1, \text{ definizione di } \mathit{fib}\ \} \\
 & x + y = \mathit{fib}(i+1) \wedge x = \mathit{fib}(i) \\
\equiv & \quad \{\ \text{sostituzione}\ \} \\
 & (x = \mathit{fib}(i+1) \wedge y = \mathit{fib}(i))^{x+y,x}_{x,y}
\end{array}
$$

Dunque il comando C' cercato è $x, y := x + y, y$. Osserviamo infine che il rafforzamento dell'invariante richiede anche di rivedere il comando di inizializzazione che, per definizione di *fib*, è semplicemente $x, y, i := 1, 0, 1$. Otteniamo così il seguente programma annotato:

$$
\begin{array}{l}
\{k \geq 1\} \\
\quad x, y, i := 1, 0, 1; \\
\{\mathsf{inv} : x = \mathit{fib}(i) \wedge y = \mathit{fib}(i-1) \wedge i \in [1, k]\}\ \{\mathsf{ft} : k - i\} \\
\quad \textbf{while}\ i \neq k\ \textbf{do} \\
\qquad \{x = \mathit{fib}(i) \wedge i \in [1, k)\} \\
\qquad x, y, i := x + y, x, i + 1 \\
\quad \textbf{endw} \\
\{x = \mathit{fib}(i) \wedge i = k\} \\
\{x = \mathit{fib}(k)\}
\end{array}
$$

9 Alcuni esercizi svolti

Esempio 9.1 Determinare un'espressione E in modo che sia soddisfatta la tripla:

$$
\begin{array}{l}
\{x \cdot y + p \cdot q = n\} \\
\quad x := x - p; \\
\quad q := E; \\
\{x \cdot y + p \cdot q = n\}
\end{array}
$$

Anziché lasciarci guidare dall'intuizione e procedere per tentativi, possiamo determinare l'espressione E in modo quasi meccanico, sfruttando la regola stessa dell'assegnamento. In base alla regola (ass) vale la tripla:

$$\{x \cdot y + p \cdot E = n\}\ q := E\ \{x \cdot y + p \cdot q = n\}$$

e quindi, utilizzando la stessa regola:

$$\{(x - p) \cdot y + p \cdot E = n\}\ x := x - p\ \{x \cdot y + p \cdot E = n\}$$

L'espressione E deve essere tale che $x \cdot y + p \cdot q = n$ implichi $(x - p) \cdot y + p \cdot E = n$. Ora, assumendo $x \cdot y + p \cdot q = n$, affinché valga $(x - p) \cdot y + p \cdot E = n$, deve essere:

$$
\begin{array}{ll}
E & = (n - (x - p) \cdot y)/p \\
 & = (x \cdot y + p \cdot q - (x - p) \cdot y)/p \\
 & = ((x - x + p) \cdot y + p \cdot q)/p \\
 & = (p \cdot y + p \cdot q)/p \\
 & = p \cdot (y + q)/p \\
 & = y + q
\end{array}
$$

Pertanto l'espressione E che soddisfa la tripla è appunto $y + q$.

Esempio 9.2 Si verifichi la tripla:

$$
\begin{array}{l}
\{k \in dom(a)\ \wedge\ \forall i : i \in dom(a)\ \wedge i \neq k.\ a[i] = 1\} \\
\quad \mathbf{if}\ a[k] = 10 \\
\quad\quad \mathbf{then}\ a[k] := a[k]\ mod\ 9 \\
\quad\quad \mathbf{else}\ a[k] := 1 \\
\quad \mathbf{fi} \\
\{\forall i \in dom(a).\ a[i] = 1\}
\end{array}
$$

dove a è una sequenza di numeri interi.

Osserviamo in primo luogo la precondizione assicura che la guardia sia definita. Infatti:

$$
\begin{array}{ll}
 & k \in dom(a)\ \wedge\ \forall i : i \in dom(a)\ \wedge\ i \neq k.\ a[i] = 1 \\
\Rightarrow & \quad \{\ \text{Sempl-}\wedge\ \} \\
 & k \in dom(a) \\
\Rightarrow & \quad \{\ \text{Definizione di } def()\ \} \\
 & def(a[k] = 10)
\end{array}
$$

e quindi la premessa della regola (if):

$$k \in dom(a)\ \wedge\ \forall i : i \in dom(a)\ \wedge\ i \neq k.\ a[i] = 1 \quad \Rightarrow \quad def(a[k] = 10)$$

è verificata. Restano da provare le altre due premesse, ovvero, le triple corrispondenti ai rami **then** ed **else**. Per quanto riguarda la prima, osserviamo che vale

$$
\begin{array}{ll}
& k \in dom(a) \ \wedge \ (\forall i : i \in dom(a) \ \wedge \ i \neq k.\ a[i] = 1) \ \wedge \ a[k] = 10 \\
\Rightarrow & \qquad \{ \text{ Sempl-}\wedge \text{ } \} \\
& (\forall i : i \in dom(a) \ \wedge \ i \neq k.\ a[i] = 1) \ \wedge \ a[k] = 10 \\
\Rightarrow & \qquad \{ \text{ Propr. di } \ mod \ \ \} \\
& \forall i : i \in dom(a) \ \wedge \ i \neq k.\ a[i] = 1 \ \wedge \ a[k] \ mod \ 9 = 1 \\
\equiv & \qquad \{ \text{ Sostituzione } \ \} \\
& (\forall i \in dom(a').\ a'[i] = 1)
\end{array}
$$

dove $a' = a\left[{}^{a[k] \ mod \ 9}/_{k}\right]$. Inoltre per la regola dell'assegnamento (ass), vale la tripla:

$$\{\forall i \in dom(a').\ a'[i] = 1\} \ a[k] := a[k] \ mod \ 9 \ \{\forall i \in dom(a).\ a[i] = 1\}$$

e quindi, per la regola (pre) si conclude che vale:

$$
\begin{array}{l}
\{k \in dom(a) \ \wedge \ (\forall i : i \in dom(a) \ \wedge \ i \neq k.\ a[i] = 1) \ \wedge \ a[k] = 10\} \\
\qquad a[k] := a[k] \ mod \ 9 \\
\{\forall i \in dom(a).\ a[i] = 1\}
\end{array}
$$

Allo stesso modo, per il ramo **else**, si prova che vale

$$
\begin{array}{l}
\{k \in dom(a) \ \wedge \ (\forall i : i \in dom(a) \ \wedge \ i \neq k.\ a[i] = 1) \ \wedge \ \neg(a[k] = 10)\} \\
\qquad a[k] := 1 \\
\{\forall i \in dom(a).\ a[i] = 1\}
\end{array}
$$

e quindi si conclude applicando la regola (if).

Esempio 9.3 Si vogliono permutare gli elementi di un array di interi a in modo che tutti gli elementi di a negativi stiano nella porzione iniziale dell'array e tutti gli elementi di a maggiori o uguali a 0 stiano nella porzione finale di a. Indichiamo con $perm(a, b)$ il fatto che gli array a, b, che si suppone abbiano lo stesso dominio, sono l'uno una permutazione dell'altro (esercizio: definire formalmente $perm(a, b)$). Il problema può essere allora specificato come segue:

$$
\begin{array}{l}
\{n \geq 0 \wedge a = A\} \\
\qquad C \\
\{perm(a, A) \wedge x \in [0, n] \wedge (\forall i \in [0, x).a[i] < 0) \wedge (\forall i \in [x, n).\ a[i] \geq 0)\},
\end{array}
$$

con n costante e a : **array** $[0, n)$ **of int**.

Trascuriamo, per il momento, il fatto che nello stato finale deve valere $perm(a, A)$ e osserviamo quanto segue.

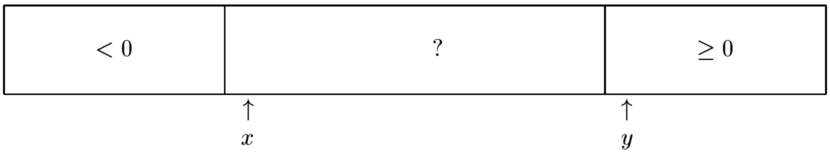
$$
\begin{array}{ll}
& x \in [0, n] \wedge (\forall i \in [0, x).a[i] < 0) \wedge (\forall i \in [x, n).\ a[i] \geq 0) \\
\Leftarrow & \qquad \{ \text{ Leibniz } \} \\
& 0 \leq x \leq y \leq n \wedge (\forall i \in [0, x).a[i] < 0) \wedge (\forall i \in [y, n).\ a[i] \geq 0) \wedge x = y
\end{array}
$$

Scegliamo allora come invariante e guardia le seguenti:

Inv: $0 \leq x \leq y \leq n \wedge (\forall i \in [0, x).a[i] < 0) \wedge (\forall i \in [y, n).\ a[i] \geq 0)$

E: $x \neq y$

Osserviamo innanzitutto che l'invariante può essere facilmente stabilita inizialmente con l'assegnamento $x, y := 0, n$ poiché i valori 0 e n per x e y banalizzano le due quantificazioni universali. Notiamo inoltre che, in uno stato che soddisfa invariante e guardia, l'array può essere rappresentato dalla seguente figura:



ovvero tutti gli elementi di a con indice in $[0, x)$ sono minori di 0 e tutti gli elementi con indice in $[y, n)$ sono non negativi. La soluzione del problema si ha, nello stato finale, quando $x = y$. Un ciclo che mantenga tale invariante e termini deve far descrescere ad ogni iterazione la distanza tra x e y, ovvero una buona funzione di terminazione è $y - x$.

Osserviamo allora quanto segue:

$$\begin{array}{ll}
 & 0 \leq x \leq y \leq n \wedge x \neq y \\
\Rightarrow & \quad \{ \text{ transitività } \} \\
 & x \in [0, n) \wedge y - 1 \in [0, n) \\
\equiv & \quad \{ \ \mathit{def}(a[x]) \equiv x \in [0, n),\ \mathit{def}(a[y-1]) \equiv y - 1 \in [0, n) \ \} \\
 & \mathit{def}(a[x]) \wedge \mathit{def}(a[y-1]).
\end{array}$$

Dunque, in uno stato in cui vale $\mathit{Inv} \wedge E$ siamo autorizzati ad ispezionare i valori $a[x]$ e $a[y-1]$. Osserviamo poi che se $a[x] < 0$ abbiamo:

$$\begin{array}{ll}
 & \mathit{Inv} \wedge E \\
\equiv & \quad \{ \text{ definizione di } \mathit{Inv} \text{ e } E \ \} \\
 & 0 \leq x < y \leq n \wedge (\forall i \in [0, x).a[i] < 0) \wedge (\forall i \in [y, n).\ a[i] \geq 0) \\
\Leftarrow & \quad \{ \ \textbf{Ip:}\ a[x] < 0, \text{ intervallo } \} \\
 & 0 \leq x + 1 \leq y \leq n \wedge (\forall i \in [0, x+1).a[i] < 0) \wedge (\forall i \in [y, n).\ a[i] \geq 0) \\
\equiv & \quad \{ \text{ sostituzione } \} \\
 & \mathit{Inv}_x^{x+1}
\end{array}$$

Dunque un comando del tipo:

$$\begin{array}{l}
\textbf{if } a[x] < 0 \\
\quad \textbf{then } x := x + 1 \\
\quad \textbf{else } C' \\
\textbf{fi}
\end{array}$$

mantiene l'invariante nel caso in cui $a[x] < 0$ ed inoltre aumenta x ovvero fa diminuire la funzione di terminazione. Per quanto riguarda C' osserviamo che se invece, $a[x] \geq 0$:

$$\begin{array}{ll}
 & \mathit{Inv} \wedge E \\
\equiv & \quad \{ \text{ definizione di } \mathit{Inv} \text{ e } E \ \} \\
 & 0 \leq x < y \leq n \wedge (\forall i \in [0, x).a[i] < 0) \wedge (\forall i \in [y, n).\ a[i] \geq 0) \\
\Leftarrow & \quad \{ \ \textbf{Ip:}\ a[x] \geq 0, a' = a[^{a[x]}/_{y-1}], \text{ intervallo } \} \\
 & 0 \leq x \leq y - 1 \leq n \wedge (\forall i \in [0, x).a'[i] < 0) \wedge (\forall i \in [y-1, n).\ a'[i] \geq 0) \\
\equiv & \quad \{ \text{ sostituzione } \} \\
 & \mathit{Inv}_{y,a}^{y-1,a'}.
\end{array}$$

Dunque un comando C' del tipo $a[y-1], y := a[x], y - 1$ mantiene di nuovo l'invariante e fa diminuire $y - x$. Tuttavia, in questo modo, non riusciamo più a garantire che l'array calcolato sia ancora una permutazione dell'array di partenza, avendo rimpiazzato l'elemento di indice $y - 1$ con quello di indice x. Osserviamo tuttavia che il comando $a[x], a[y-1], y := a[y-1], a[x], y - 1$, continua a mantenere l'invariante e soddisfa anche la tripla

$$\{\mathit{perm}(a, A)\}\ a[x], a[y-1], y := a[y-1], a[x], y - 1\ \{\mathit{perm}(a, A)\}.$$

Concludiamo che una soluzione del problema è data dal seguente programma annotato:

$\{a = A\}$
$\quad x, y := 0, n;$
$\{\mathsf{inv} : \mathit{Inv}\}\ \{\mathsf{ft} : y - x\}$
$\quad$ **while** $x \neq y$ **do**
$\qquad \{\mathit{Inv} \wedge x \neq y\}$
$\qquad$ **if** $a[x] < 0$
$\qquad\quad$ **then** $x := x + 1$
$\qquad\quad$ **else** $a[x], a[y-1], y := a[y-1], a[x], y - 1$
$\qquad$ **fi**
$\quad$ **endw**
$\{\mathit{Inv} \wedge x = y\}$
$\{x \in [0, n[\wedge (\forall i \in [0, x).\ a[i] < 0) \wedge (\forall i \in [x, n).\ a[i] \geq 0) \wedge \mathit{perm}(a, A)\}$

dove:

$$\mathit{Inv} : \mathit{perm}(a, A) \wedge 0 \leq x \leq y \leq n \ \wedge \ (\forall i \in [0, x).\ a[i] < 0) \wedge (\forall i \in [y, n).\ a[y] \geq 0)$$

Esempio 9.4 Consideriamo il problema di calcolare il massimo comun divisore tra due valori naturali. Il problema può essere specificato come segue

$$\{a > 0 \wedge b > 0\}\ C\ \{x = \mathit{mcd}(a, b)\} \qquad \text{con } a, b \text{ costanti.}$$

Ricordiamo che la funzione mcd soddisfa le seguenti proprietà, con $x, y > 0$:

(i) $\mathit{mcd}(x, x) = x$

(ii) $\mathit{mcd}(x, y) = \mathit{mcd}(x - y, y) \qquad$ se $x > y$

(iii) $\mathit{mcd}(x, y) = \mathit{mcd}(x, y - x) \qquad$ se $x < y$.

È facile convincersi che il seguente frammento di programma risolve il problema dato:

$\{a > 0 \wedge b > 0\}$
$\quad x, y := a, b;$
$\{\mathsf{inv} :?\}\ \{\mathsf{ft} :?\}$
$\quad$ **while** $x \neq y$ **do**
$\qquad$ **if** $x > y$
$\qquad\quad$ **then** $x := x - y$
$\qquad\quad$ **else** $y := y - x$
$\qquad$ **fi**
$\quad$ **endw**
$\{x = \mathit{mcd}(a, b)\}$

Per dimostrare la correttezza del frammento di programma, dobbiamo però individuare un'invariante ed una funzione di terminazione per il ciclo. L'asserzione finale può essere manipolata come segue:

$$
\begin{array}{ll}
 & x = \mathit{mcd}(a, b) \\
\equiv & \quad \{\ \text{proprietà (i)}\ \} \\
 & \mathit{mcd}(x, x) = \mathit{mcd}(a, b) \\
\Leftarrow & \quad \{\ \text{Leibniz}\ \} \\
 & \mathit{mcd}(x, y) = \mathit{mcd}(a, b) \wedge x = y
\end{array}
$$

Osservando che $x = y$ è proprio la guardia del ciclo negata, consideriamo come invariante candidata l'asserzione:

$$\mathit{mcd}(x, y) = \mathit{mcd}(a, b).$$

Osserviamo innanzitutto che il comando di assegnamento che precede il ciclo rende vera l'invariante all'inizio. Se rafforziamo l'invariante con la congiunzione $x > 0 \wedge y > 0$, ottenendo così

$Inv : x > 0 \wedge y > 0 \wedge mcd(x, y) = mcd(a, b)$

la correttezza del ciclo discende immediatamente dalle proprietà (ii) e (iii) date sopra. Per quanto concerne la funzione di terminazione, possiamo scegliere, ad esempio, $x + y$. Lasciamo per esercizio la verifica completa del programma annotato così ottenuto.

$\{a > 0 \wedge b > 0\}$
  $x, y := a, b;$
$\{\mathsf{inv} : x > 0 \wedge y > 0 \wedge mcd(x, y) = mcd(a, b)\}$ $\{\mathsf{ft} : x + y\}$
  while $x \neq y$ **do**
  $\{x > 0 \wedge y > 0 \wedge mcd(x, y) = mcd(a, b) \wedge x \neq y\}$
    if $x > y$
      then $x := x - y$
      else $y := y - x$
    fi
  endw
$\{x = mcd(a, b)\}$

L'esempio mette in luce un'ulteriore tecnica per determinare l'invariante di un ciclo, simile a quella descritta nell'Esempio 8.2. Data una *post* del tipo $P(a_1, \ldots, a_n)$, con $a_1, \ldots, a_n$ costanti del problema, si rimpiazzano tali costanti con altrettante variabili $x_1, \ldots, x_n$ ottenenedo come invariante candidata un'uguaglianza del tipo $P(a_1, \ldots, a_n) = P(x_1, \ldots, x_n)$. La guardia candidata dipende poi dal particolare problema in esame e dalle proprietà di P. Inoltre, può rendersi necessario un rafforzamento dell'invariante (come nel caso del programma precedente, in cui l'invariante candidata $mcd(x, y) = mcd(a, b)$ è stata rafforzata con la congiunzione $x > 0 \wedge y > 0$, per garantire la buona definizione delle altre condizioni e della guardia. Si noti che l'uguaglianza $P(a_1, \ldots, a_n) = P(x_1, \ldots, x_n)$ può essere facilmente stabilita all'inizio con l'assegnamento $x_1, \ldots, x_n := a_1, \ldots, a_n$.

Esempio 9.5 Sia f la successione definita come segue:

$$f(k) = \begin{cases} 0 & \text{se } k = 0 \\ 1 & \text{se } k = 1 \\ f(k/2) & \text{se } k \geq 2 \ \wedge \ pari(k) \\ f((k-1)/2) + f((k+1)/2) & \text{se } k \geq 2 \ \wedge \ dispari(k) \end{cases}$$

Si vuole determinare l'n-esimo termine di tale successione, con n costante naturale. Il problema può essere specificato mediante la tripla:

$\{n \geq 0\}$ C $\{h = f(n)\}$.

Il tentativo di operare come nel caso dell'esempio 8.5 si rivela subito inconcludente. Se consideriamo infatti l'invariante

$h = f(i) \wedge i \in [0, n]$

insieme con la guardia $i \neq n$, osserviamo che:

- è possibile inizializzare facilmente h e i in modo da stabilire l'invariante stessa (mediante l'assegnamento $h, i := 0, 0$);
- è invece problematico mantenere l'invariante nel corpo del potenziale ciclo.

Intuitivamente, per aumentare il valore di i e al tempo stesso mantenere l'invariante, bisognerebbe rafforzare quest'ultima in modo da mantenere, ad ogni passo, anche un'uguaglianza del tipo $y = f(i \ div \ 2)$, ed una del tipo $f(i \ div \ 2 + 1)$ (si confronti questa situazione con l'esempio 8.5, in cui si è dovuto rafforzare l'invariante candidata con una asserzione del tipo $y = fib(i - 1)$). Ma è proprio tale rafforzamento dell'invariante che rende problematico mantenerla (omettiamo i dettagli per brevità, invitando chi legge ad analizzarli per esercizio).

Una attenta analisi della successione f porta tuttavia a convincersi del fatto che, qualunque sia n, è sempre possibile esprimere il valore $f(n)$ come una combinazione lineare del tipo $h \cdot f(x) + k \cdot f(x+1)$, con $x \in [0, n]$. Per convincerci di questo fatto, calcoliamo, ad esempio, il valore $f(19)$. Abbiamo:

$$
\begin{array}{ll}
 & f(19) \\
= & \qquad \{ \text{ definizione di } f \ \} \\
 & f(9) + f(10) \qquad\qquad\qquad (*) \\
= & \qquad \{ \text{ definizione di } f \ \} \\
 & f(4) + f(5) + f(5) \\
= & \qquad \{ \text{ calcolo } \} \\
 & f(4) + 2 \cdot f(5) \qquad\qquad\quad (*) \\
= & \qquad \{ \text{ definizione di } f \ \} \\
 & f(2) + 2 \cdot (f(2) + f(3)) \\
= & \qquad \{ \text{ calcolo } \} \\
 & 3 \cdot f(2) + 2 \cdot f(3)) \qquad\qquad (*) \\
= & \qquad \{ \text{ definizione di } f \ \} \\
 & 3 \cdot f(1) + 2 \cdot (f(1) + f(2)) \\
= & \qquad \{ \text{ calcolo } \} \\
 & 5 \cdot f(1) + 2 \cdot f(2) \qquad\qquad (*) \\
= & \qquad \{ \text{ definizione di } f \ \} \\
 & 5 \cdot f(1) + 2 \cdot f(1) \\
= & \qquad \{ \text{ calcolo } \} \\
 & 7 \cdot f(1) \qquad\qquad\qquad\qquad (*) \\
= & \qquad \{ \text{ definizione di } f \ \} \\
 & 7
\end{array}
$$

Tutte le espressioni contrassegnate da (*) indicano che il valore di $f(19)$ è esprimibile come $h \cdot f(x) + k \cdot f(x+1)$ per valori *decrescenti* di x. Anche l'ultimo termine contrassegnato da (*) è di questo tipo, osservando che

$$
\begin{array}{ll}
 & 7 \cdot f(1) \\
= & \qquad \{ \ f(0) = 0 \ \} \\
 & k \cdot f(0) + 7 \cdot f(1)
\end{array}
$$

per qualsiasi k.

Queste osservazioni conducono a determinare come invariante e guardia candidate le seguenti:

Inv: $x \in [0, n] \wedge f(n) = k \cdot f(x) + h \cdot f(x+1)$

E: $x \neq 0$.

Osserviamo che:

$$
\begin{array}{ll}
 & x \in [0, n] \wedge f(n) = k \cdot f(x) + h \cdot f(x+1) \wedge x = 0 \\
\Rightarrow & \qquad \{ \text{ Leibniz } \} \\
 & f(n) = k \cdot f(0) + h \cdot f(1) \\
\equiv & \qquad \{ \text{ definizione di } f \ \} \\
 & f(n) = h
\end{array}
$$

Dunque un ciclo che mantenga *Inv* come invariante e che sia governato dalla guardia E permette di risolvere correttamente il problema. Osserviamo inoltre che l'invariante può essere stabilita facilmente all'inizio del ciclo mediante l'assegnamento $x, h, k := n, 0, 1$.

Abbiamo dunque un ciclo candidato del tipo:

$\{n \geq 0\}$
$\quad x, h, k := n, 0, 1;$
$\{\mathsf{inv} : x \in [0, n] \wedge f(n) = k \cdot f(x) + h \cdot f(x+1)\}$ $\{\mathsf{ft} : ?\}$
$\quad$ **while** $x \neq 0$ **do**
$\quad\quad \{x \in (0, n] \wedge f(n) = k \cdot f(x) + h \cdot f(x+1)\}$
$\quad\quad$ C
$\quad$ **endw**
$\{h = f(n)\}$

Osservando che il comando C deve portare in uno stato in cui $x = 0$, ovvero far decrescere il valore di x, possiamo scegliere x stessa come funzione di terminazione. Consideriamo ora la tripla che C deve soddisfare per mantenere l'invariante, ovvero:

$$\{x \in (0, n] \wedge f(n) = k \cdot f(x) + h \cdot f(x+1)\}\ C\ \{x \in [0, n] \wedge f(n) = k \cdot f(x) + h \cdot f(x+1)\}.$$

Osserviamo che, se x é pari:

$$
\begin{array}{ll}
 & k \cdot f(x) + h \cdot f(x+1) \\
= & \quad \{\ \mathbf{Ip:}\ pari(x),\ \text{definizione di } f\ \} \\
 & k \cdot f(x\ div\ 2) + h \cdot f(x+1) \\
= & \quad \{\ pari(x)\ \Rightarrow\ dispari(x+1)\ \} \\
 & k \cdot f(x\ div\ 2) + h \cdot (f(x\ div\ 2) + f(x\ div\ 2 + 1)) \\
= & \quad \{\ \text{calcolo}\ \} \\
 & (h+k) \cdot f(x\ div\ 2) + h \cdot f(x\ div\ 2 + 1) \\
= & \quad \{\ \text{sostituzione}\ \} \\
 & (k \cdot f(x) + h \cdot f(x+1))_{k,x}^{h+k,x\ div\ 2}
\end{array}
$$

Dunque, nell'ipotesi $pari(x)$ un assegnamento del tipo $k, x := h + k, x\ div\ 2$ mantiene l'invariante. Osserviamo inoltre che, poiché la guardia del ciclo assicura $x \neq 0$, tale assegnamento fa decrescere propriamente x e dunque la funzione di terminazione. Supponiamo ora x dispari. Abbiamo:

$$
\begin{array}{ll}
 & k \cdot f(x) + h \cdot f(x+1) \\
= & \quad \{\ \mathbf{Ip:}\ dispari(x),\ \text{definizione di } f\ \} \\
 & k \cdot (f(x\ div\ 2) + f(x\ div\ 2 + 1)) + h \cdot f(x+1) \\
= & \quad \{\ dispari(x)\ \Rightarrow\ pari(x+1)\ \} \\
 & k \cdot (f(x\ div\ 2) + f(x\ div\ 2 + 1)) + h \cdot f((x+1)\ div\ 2) \\
= & \quad \{\ dispari(x)\ \Rightarrow\ (x+1)\ div\ 2 = x\ div\ 2 + 1\ \} \\
 & k \cdot (f(x\ div\ 2) + f(x\ div\ 2 + 1)) + h \cdot f(x\ div\ 2 + 1) \\
= & \quad \{\ \text{calcolo}\ \} \\
 & k \cdot f(x\ div\ 2) + (h+k) \cdot f(x\ div\ 2 + 1) \\
= & \quad \{\ \text{sostituzione}\ \} \\
 & (k \cdot f(x) + h \cdot f(x+1))_{h,x}^{h+k,x\ div\ 2}
\end{array}
$$

Dunque, nell'ipotesi $dispari(x)$ un assegnamento del tipo $h, x := h + k, x\ div\ 2$ mantiene l'invariante. Abbiamo, in cocnlusione, che il seguente frammento di programma risolve il problema dato:

$\{n \geq 0\}$
$\quad x, h, k := n, 0, 1;$
$\{\mathsf{inv} : x \in [0, n] \wedge f(n) = k \cdot f(x) + h \cdot f(x+1)\}$ $\{\mathsf{ft} : x\}$
$\quad$ **while** $x \neq 0$ **do**
$\quad\quad \{x \in (0, n] \wedge f(n) = k \cdot f(x) + h \cdot f(x+1)\}$
$\quad\quad$ **if** $x\ mod\ 2 = 0$
$\quad\quad\quad$ **then** $k, x := h + k, x\ div\ 2$
$\quad\quad\quad$ **else** $h, x := h + k, x\ div\ 2$
$\quad\quad$ **fi**
$\quad$ **endw**
$\{h = f(n)\}$

Esempio 9.6 Sia g una funzione dall'intervallo non vuoto $[a, b)$ di $\mathbb{N}$ in $\mathbb{Z}$. Supponendo di rappresentare g mediante una sequenza, si vuole determinare l'elemento minimo del dominio di g in cui g assume il suo valore massimo. Specificare il problema come tripla di Hoare, risolvere il problema e dimostrare la correttezza della soluzione proposta.

Una possibile specifica del problema è:

$$\begin{array}{l} \{pre:\ a < b+1\} \\ \quad \mathrm{C} \\ \{post:\ k \in [a,b)\ \wedge\ \forall i \in [a,b).\ g[i] \leq g[k]\ \wedge\ \forall i \in [a,k).\ g[i] < g[k]\} \end{array}$$

dove g : **array** $[a, b)$ **of integer** costante.

La precondizione $\{a < b+1\}$ formalizza il fatto che l'intervallo $[a, b)$ è non vuoto. Per quanto riguarda la postcondzione, la variabile $k \in [a, b)$ contiene il punto del dominio cercato, infatti:

- $\forall i \in [a,b).\ g[i] \leq g[k]$ asserisce che $g[k]$ è il valore massimo assunto da g,
- $\forall i \in [a,k).\ g[i] < g[k]$ asserisce che k è il minimo elemento del dominio sul quale g assume il valore massimo, $g[k]$.

Il problema sarà certamente risolto mediante un costrutto iterativo che analizza la sequenza, cercando l'elemento del dominio che soddisfa le condizioni desiderate. Per determinare l'invariante del ciclo un'idea consiste nell'applicare una delle tecniche introdotte Paragrafo 8, ovvero il rimpiazzamento di una costante con una variabile nell'asserzione finale. Sostituiamo quindi b con una variabile x, ottenendo, come candidato per l'invariante:

$$Inv\colon\ x \in [a,b]\ \wedge\ k \in [a,x)\ \wedge\ \forall i \in [a,x).\ g[i] \leq g[k]\ \wedge\ \forall i \in [a,k).\ g[i] < g[k]$$

Un modo per stabilire inizialmente tale asserzione consiste nell'assegnare alla variabile k il valore a ed a x il valore $a+1$, dato che cosí l'intervallo $[a, x)$ ha un solo elemento e $[a, k)$ è vuoto. L'idea è qundi quella di scorrere tutta la sequenza con la variabile x, dall'inizio alla fine, incrementandola ad ogni iterazione. Scegliamo dunque come guardia del ciclo

$$G\ :\ x \neq b$$

e una buona funzione di terminazione sarà $b-x$. Il frammento di programma finora delineato è il seguente.

$$\begin{array}{l} \{x < y+1\} \\ \quad x, k := a+1, a; \\ \{\mathsf{inv}: x \in [a,b]\ \wedge\ k \in [a,x)\ \wedge\ \forall i \in [a,x).\ g[i] \leq g[k]\ \wedge\ \forall i \in [a,k).\ g[i] < g[k]\}\ \{\mathsf{ft}: b-x\} \\ \quad \textbf{while}\ x \neq b\ \textbf{do} \\ \qquad \mathrm{C}; \\ \qquad x := x+1 \\ \quad \textbf{endw} \\ \{k \in [a,b)\ \wedge\ \forall i \in [a,b).\ g[i] \leq g[k]\ \wedge\ \forall i \in [a,k).\ g[i] < g[k]\} \end{array}$$

Per determinare il comando C osserviamo che, se $g[x] > g[k]$ allora

$$\begin{array}{ll} & Inv \wedge G \\ \equiv & \quad \{\ \text{Definizione}\ \} \\ & x \in [a,b]\ \wedge\ k \in [a,x)\ \wedge\ \forall i \in [a,x).\ g[i] \leq g[k]\ \wedge\ \forall i \in [a,k).\ g[i] < g[k]\ \wedge\ x \neq b \\ \Rightarrow & \quad \{\ x \in [a,b]\ \wedge\ x \neq b\ \Rightarrow\ x \in [a,b-1]\ \} \\ & x \in [a,b-1]\ \wedge\ k \in [a,x)\ \wedge\ \forall i \in [a,x).\ g[i] \leq g[k]\ \wedge\ \forall i \in [a,k).\ g[i] < g[k] \\ \equiv & \quad \{\ x \in [a,b-1]\ \Rightarrow\ x+1 \in [a,b]\ \} \\ & x+1 \in [a,b]\ \wedge\ k \in [a,x)\ \wedge\ \forall i \in [a,x).\ g[i] \leq g[k]\ \wedge\ \forall i \in [a,k).\ g[i] < g[k] \\ \Rightarrow & \quad \{\ \text{Intervallo}\ \} \\ & x+1 \in [a,b]\ \wedge\ x \in [a,x+1)\ \wedge\ \forall i \in [a,x).\ g[i] \leq g[k]\ \wedge\ \forall i \in [a,k).\ g[i] < g[k] \\ \Rightarrow & \quad \{\ \text{Sempl-}\wedge\ \} \end{array}$$

$$
\begin{array}{ll}
& x+1 \in [a,b] \ \wedge \ x \in [a,x+1) \ \wedge \ \forall i \in [a,x).\ g[i] \leq g[k] \\
\Rightarrow & \quad \{ \ \textbf{Ip:}\ g[x] > g[k], \text{ transitività e riflessività di } \leq \ \} \\
& x+1 \in [a,b] \ \wedge \ x \in [a,x+1) \ \wedge \ \forall i \in [a,x+1).\ g[i] \leq g[x] \ \wedge \ \forall i \in [a,x).\ g[i] < g[x] \\
\equiv & \quad \{ \text{ Sostituzione } \} \\
& \mathit{Inv}_{k,x}^{x,x+1}
\end{array}
$$

Dunque la regola dell'assegnamento assicura che vale la tripla

$$\{\mathit{Inv} \ \wedge \ G\} \ k,x := x,x+1 \ \{\mathit{Inv}\}$$

È facile dimostrare poi che se $g[x] \leq g[k]$, allora vale

$$\mathit{Inv} \ \wedge \ G \ \equiv \ \mathit{Inv}_x^{x+1}$$

e quindi la tripla $\{\mathit{Inv} \ \wedge \ G\}$ $x := x+1$ $\{\mathit{Inv}\}$ è corretta.

Queste considerazioni ci permettono di concludere, utilizzando la regola del condizionale, che vale la tripla:

$\{\mathit{Inv} \ \wedge \ G\}$
    if $g[x] > g[k]$
        then $k,x := x,x+1$
        else $x := x+1$
    fi
$\{\mathit{Inv}\}$

e quindi anche:

$\{\mathit{Inv} \ \wedge \ G\}$
    if $g[x] > g[k]$
        then $k := x$
    fi
    $x := x+1$
$\{\mathit{Inv}\}$

come il lettore può facilmente verificare per esercizio.

Possiamo dunque concludere che il programma desiderato è

$\{x < y+1\}$
    $x,k := a+1, a;$
$\{\mathsf{inv} : x \in [a,b] \ \wedge \ k \in [a,x) \ \wedge \ \forall i \in [a,x).\ g[i] \leq g[k] \ \wedge \ \forall i \in [a,i).\ g[i] < g[k]\}$ $\{\mathsf{ft} : b - x\}$
    while $x \neq b$ **do**
        if $g[x] > g[k]$
            then $k := x$
        fi
        $x := x+1$
    endw
$\{k \in [a,b) \ \wedge \ \forall i \in [a,b).\ g[i] \leq g[k] \ \wedge \ \forall i \in [a,k).\ g[i] < g[k]\}$

La dimostrazione delle ipotesi di progresso e terminazione è banale, così com'è semplice verificare che $\mathit{Inv} \wedge \neg G \ \Rightarrow \ \mathit{post}$.

Esempio 9.7 Siano $m,n > 0$ due costanti e siano a,b due sequenze di interi con dominio $[0,m)$ e $[0,n)$ rispettivamente. Si determini un comando C, che non modifica a e b e che soddisfa la seguente tripla di Hoare:

$\{pre: \ \exists k \in [0,n).\ b[k] \neq 0\}$
C
$\{post: \ (\neg P \ \wedge \ x = \mathbf{min}\{i \in [0,n) : b[i] \neq 0\}) \ \vee \ (P \ \wedge \ x = \mathbf{min}\{i \in [0,m) : a[i] \neq 0\})\}$

dove P è la seguente asserzione:

$$P \equiv (\exists k \in [0,m).\ a[k] \neq 0).$$

L'interpretazione informale dell'asserzione finale è la seguente: se in a ci sono elementi non nulli, allora x deve essere il minimo elemento del dominio di a tale che $a[x] \neq 0$, altrimenti x deve essere il minimo elemento del dominio di b tale che $b[x] \neq 0$. L'asserzione iniziale *pre* assicura che in b ci sono elementi non nulli. L'idea è quindi quella di applicare lo schema di ricerca lineare incerta alla sequenza a. Se si trova un elemento non nullo, abbiamo finito, altrimenti si applica lo schema di ricerca lineare certa alla sequenza b.

Formalmente, lo schema di ricerca lineare incerta applicato alla ricerca di un elemento non nullo in a diviene il seguente comando C_1:

$\{pre_1\}$
  $x, trovato := 0, false;$
$\{\mathsf{inv}: Inv_1\}\ \{\mathsf{ft}: m - (x + int(trovato))\}$
  while $x < m$ *and not trovato* **do**
    if $a[x] \neq 0$
      then $trovato := true$
      else $x := x + 1$
    fi
  endw
$\{post_1\}$

dove la precondizione, pre_1: $\forall x \in [0,m).def(a[x])) \equiv \mathbf{tt}$. La postcondizione $post_1$ si istanzia come segue:

$$x \in [0,m] \ \wedge \ (\forall j \in [0,x).\ a[j] = 0) \ \wedge \ (x < m \ \Rightarrow \ a[x] \neq 0).$$

Ora, se $x < m$, abbiamo:

$$\begin{array}{ll}
 & post_1 \\
\Rightarrow & \quad \{ \ \mathbf{Ip:}\ x < m \ \} \\
 & x \in [0,m) \ \wedge \ (\forall j \in [0,x).\ a[j] = 0) \ \wedge \ a[x] \neq 0 \\
\Rightarrow & \quad \{ \text{ Intro-}\exists \ \} \\
 & x \in [0,m) \ \wedge \ (\forall j \in [0,x).\ a[j] = 0) \ \wedge \ a[x] \neq 0 \ \wedge \ \exists k \in [0,m).\ a[k] \neq 0 \\
\equiv & \quad \{ \text{ Definizione di } P \ \} \\
 & x \in [0,m) \ \wedge \ (\forall j \in [0,x).\ a[j] = 0) \ \wedge \ a[x] \neq 0 \ \wedge \ P \\
\equiv & \quad \{ \text{ Definizione di } \mathbf{min} \ \} \\
 & x = \mathbf{min}\{k \in [0,m) : a[k] \neq 0\} \ \wedge \ P
\end{array}$$

Se invece $x = m$, allora è facile provare che $post_1$ implica $(\neg P \ \wedge \ x = m)$. Dunque riassumendo:

$$post_1 \ \Rightarrow \ (x < m \ \wedge \ P \wedge \ x = \mathbf{min}\{k \in [0,m) : a[k] \neq 0\}) \ \vee \ (x = m \ \wedge \ \neg P).$$

e quindi, per le regole *(post)* e *(invariante)*, vale la tripla:

$$\{pre\}\ C_1\ \{((x < m \ \wedge \ P \wedge \ x = \mathbf{min}\{k \in [0,m) : a[k] \neq 0\}) \ \vee \ (x = m \ \wedge \ \neg P)) \ \wedge \ pre\}.$$

Lo schema di ricerca lineare certa applicato a b per la ricerca del minimo elemento x del dominio della sequenza tale che $b[x] \neq 0$, è il seguente comando C_2:

$\{pre_2\}$
$\quad x := 0;$
$\{\mathsf{inv} : Inv_2\}\ \{\mathsf{ft} : n - x\}$
$\quad$ **while** $b[x] \neq 0$ **do**
$\qquad x := x + 1$
$\quad$ **endw**
$\{post_2\}$

dove la precondizione $pre_2 \ \equiv\ I \in [0, n) \ \wedge\ b[I] \neq 0 \ \wedge\ (x \in [0, n) \Rightarrow def(b[x] = 0))$ è implicata dalla precondizione *pre* del problema generale. La postcondizione, $post_2$:

$$x \in [0, n) \ \wedge\ (\forall j \in [0, x).\ b[j] = 0) \ \wedge\ b[x] \neq 0,$$

per definizione di **min**, è equivalente a $x = \mathbf{min}\{k \in [0, n) : b[k] \neq 0\}$.

Utilizzando la regola (if) per il condizionale si prova facilmente che è corretta la seguente tripla:

$\{((x < m \ \wedge\ P \ \wedge\ x = \mathbf{min}\{k \in [0, m) : a[k] \neq 0\}) \ \vee\ (x = m \ \wedge\ \neg P)) \wedge\ pre\}$
$\quad$ **if** $x = m$ **then**
$\qquad C_2$
$\quad$ **fi**
$\{post :\ (\neg P \ \wedge\ x = \mathbf{min}\{i \in [0, n) : b[i] \neq 0\}) \ \vee\ (P \ \wedge\ x = \mathbf{min}\{i \in [0, m) : a[i] \neq 0\})\}$

Infatti, consideriamo il caso $x \neq m$:

$$\begin{array}{ll}
 & ((x < m \ \wedge\ P \ \wedge\ x = \mathbf{min}\{k \in [0, m) : a[k] \neq 0\}) \ \vee\ (x = m \ \wedge\ \neg P)) \wedge\ pre \ \wedge\ x \neq m \\
\Rightarrow & \quad \{ \text{ Sempl-}\wedge \ \} \\
 & ((x < m \ \wedge\ P \ \wedge\ x = \mathbf{min}\{k \in [0, m) : a[k] \neq 0\}) \ \vee\ (x = m \ \wedge\ \neg P)) \ \wedge\ x \neq m \\
\equiv & \quad \{ \text{ Distrib. di } \wedge \text{ rispetto a } \vee \ \} \\
 & (x < m \ \wedge\ x \neq m\ P \ \wedge\ x = \mathbf{min}\{k \in [0, m) : a[k] \neq 0\}) \ \vee\ (x = m \ \wedge\ x \neq m \ \wedge\ \neg P) \\
\Rightarrow & \quad \{ \text{ Sempl-}\wedge \text{ e } (x = m \ \wedge\ x \neq m) \ \equiv\ \mathbf{ff} \ \} \\
 & (P \ \wedge\ x = \mathbf{min}\{k \in [0, m) : a[k] \neq 0\}) \ \vee\ (\mathbf{ff} \wedge\ \neg P) \\
\equiv & \quad \{ \text{ Zero } \} \\
 & (P \ \wedge\ x = \mathbf{min}\{k \in [0, m) : a[k] \neq 0\}) \ \vee\ \mathbf{ff} \\
\equiv & \quad \{ \text{ Identità } \} \\
 & (P \ \wedge\ x = \mathbf{min}\{k \in [0, m) : a[k] \neq 0\}) \\
\equiv & \quad \{ \text{ Intro-}\vee \text{ e Definizione di } post \ \} \\
 & post
\end{array}$$

Nel caso in cui, invece, $x = m$, possiamo provare, con passaggi simili a quelli appena visti, che:

$$((x < m \ \wedge\ P \ \wedge\ x = \mathbf{min}\{k \in [0, m) : a[k] \neq 0\}) \ \vee\ (x = m \ \wedge\ \neg P)) \wedge\ pre \ \wedge\ x \neq m \quad \Rightarrow \quad pre \ \wedge\ \neg P.$$

e per la regola *(invariante)* si conclude la correttezza della tripla:

$\{pre \ \wedge\ \neg P\}$
$\quad C_2$
$\{post_2 \ \wedge\ \neg P\}$

Dato che $post_2 \ \wedge\ \neg P \quad \Rightarrow \quad post$, per la regola $(post)$ vale anche la tripla:

$\{pre \ \wedge\ \neg P\}$
$\quad C_2$
$\{post\}$

A questo punto è immediato, utilizzando le considerazioni fatte finora, concludere la correttezza di:

$\{pre : \ \exists k \in [0,n).\ b[k] \neq 0\}$
$\quad x, trovato := 0, false;$
$\{\mathsf{inv} : Inv_1\}\ \{\mathsf{ft} : m - (x + int(trovato))\}$
$\quad$ **while** $x < m$ *and not* $trovato$ **do**
$\qquad$ **if** $a[x] \neq 0$
$\qquad\quad$ **then** $trovato := true$
$\qquad\quad$ **else** $x := x + 1$
$\qquad$ **fi**
$\quad$ **endw**
$\{(P \ \wedge \ x = \mathbf{min}\{k \in [0,m) : a[k] \neq 0\}) \ \vee \ (\neg P \ \wedge \ x = m)\}$
$\quad$ **if** $x = m$ **then**
$\qquad x := 0;$
$\quad \{\mathsf{inv} : Inv_2\}\ \{\mathsf{ft} : n - x\}$
$\qquad$ **while** $b[x] = 0$ **do**
$\qquad\quad x := x + 1$
$\qquad$ **endw**
$\quad$ **fi**
$\{post : \ (\neg P \ \wedge \ x = \mathbf{min}\{i \in [0,n) : b[i] \neq 0\}) \ \vee \ (P \ \wedge \ x = \mathbf{min}\{i \in [0,m) : a[i] \neq 0\})\}$

Esempio 9.8 Determinare il comando C in modo che sia soddisfatta la seguente tripla di Hoare:

$$\{dom(a) = [0,n) \ \wedge \ k \in [0,n)\}\ C\ \{a[k] = \mathbf{max}\{a[i] : i \in [0,n)\}$$

dove a : **array** $[0,n)$ **of integer**, k e n sono costanti naturali e C contiene al più un ciclo.

A prima vista questo potrebbe sembrare un problema di ricerca di massimo nella sequenza a. Tuttavia si deve subito notare che k è una costante, per cui l'unico modo per soddisfare la tripla è far si che il comando modifichi opportunamente la sequenza. Un'idea che sorge naturalmente è quella di assegnare ad $a[k]$ il massimo tra gli elementi di a. Tuttavia questa soluzione richiede di individuare di tale massimo ed è inutilmente complicata. Il problema è che siamo fuorviati dal tentativo di apportare il minor numero di modifiche possibile alla sequenza data a, quando invece la specifica non richiede nulla in questo senso!

Una semplice soluzione dell'esercizio consiste nell'azzerare tutti gli elementi di a, con l'esclusione di $a[k]$ che viene posto al valore 1. Si ottiene così il comando:

$\{dom(a) = [0,n) \ \wedge \ k \in [0,n)\}$
$\quad x := 0;$
$\{\mathsf{inv} : \forall i \in [0,x).\ a[i] = 0\}\ \{\mathsf{ft} : n - x\}$
$\quad$ **while** $x < n$ **do**
$\qquad a[x] := 0;$
$\qquad x := x + 1$
$\quad$ **endw**;
$\{\forall i \in [0,n).\ a[i] = 0\}$
$\quad a[k] := 1$
$\{\forall i \in [0,n).\ ((i \neq k \ \Rightarrow \ a[i] = 0) \ \wedge \ (i = k \ \Rightarrow \ a[i] = 1))\}$
$\{a[k] = \mathbf{max}\{a[i] : i \in [0,n)\}$

La verifica di correttezza è banale ed è lasciata al lettore come esercizio.

Esempio 9.9 Un *picco* è una sequenza di naturali *distinti* in cui esiste un elemento, detto *culmine*, tale che:

1. la sequenza composta dagli elementi che precedono il culmine e dal culmine stesso è crescente;
2. la sequenza composta dal culmine stesso e dagli elementi che lo seguono è decrescente.

Sia a una sequenza di numeri interi con dominio $[0, n)$, con $n \geq 1$, tale che i suoi elementi formano un picco. Dare un'asserzione che formalizzi le proprietà della sequenza e applicare lo schema di ricerca binaria per cercare il culmine in a. Si espliciti la proprietà utilizzata nella ricerca binaria, ricordando che il culmine può essere anche il primo o l'ultimo elemento della sequenza.

L'asserzione che formalizza l'essere un picco per un array a può essere la seguente:

$$P(a) \;\equiv\; I \in [0,n) \;\wedge\; (\forall 0 \leq i < j \leq I.\ a[i] < a[j]) \;\wedge\; (\forall I \leq i < j < n.\ a[i] > a[j]),$$

dove la variabile di specifica I rappresenta il culmine della sequenza.

Dato che abbiamo la certezza che la sequenza è un picco, la ricerca del culmine può fondarsi su una semplice considerazione: dato un elemento x del dominio della sequenza, $x \in [1, n)$:

- se $a[x-1] < a[x]$ siamo nella parte crescente della sequenza e quindi il culmine segue x;
- se $a[x-1] > a[x]$ siamo nella parte decrescente della sequenza e quindi il culmine precede x .

Per applicare lo schema di ricerca binaria possiamo dunque utilizzare come proprietà:

$$E(x) \;\equiv\; a[x-1] < a[x], \qquad \text{per } x \in [1, n).$$

È facile dimostrare che vale $ord(a, E)$: un picco è ordinato rispetto alla proprietà $E(x)$, ovvero per ogni $x \in [1, n)$:

- se $E(x)$ allora $E(i)$ per ogni $i \in [1, x]$;
- se $\neg E(x)$ allora $\neg E(i)$ per ogni $i \in [x, n)$.

Si noti che la ricerca avviene in $[1, n)$ dato che la proprietà coinvolge una coppia di elementi: $a[x]$ ed il suo predecessore. Lo schema generale si istanzia come segue:

```
{pre}
   x, y := 0, n;
{inv : Inv} {ft : y − x}
   while y ≠ x + 1 do
      m := (x + y) div 2;
      if a[x − 1] < a[x]
         then x := m
         else y := m
      fi
   endw
{post}
```

dove la precondizione:

$$pre\text{: } ord(a, E) \;\wedge\; (x \in [1, n) \Rightarrow def(E(x)))$$

in base alle precedenti considerazioni è soddisfatta. La postcondizione si istanzia come segue:

$$\begin{aligned}&(x = 0 \;\wedge\; (\forall i \in [1, n).\ a[i-1] > a[i])) \vee \\ &(x \in [1, n) \;\wedge\; (\forall i \in [1, x).\ a[i-1] < a[i]) \wedge\; (\forall i \in [x, n).\ a[i-1] > a[i])).\end{aligned}$$

ovvero asserisce esattamente che x è il culmine del picco: se $x = 0$ la sequenza è decrescente e quindi il culmine è proprio $a[0]$; se invece $x \in [1, n)$ la sequenza cresce fino ad x e poi decresce, ovvero, ancora $a[x]$ è il culmine.

Esempio 9.10 Utilizzando uno schema di programma noto si definisca un comando C che verifichi la seguente tripla di Hoare:

$$\{dom(a) = [0,n) \ \wedge \ n \geq 0\}$$
$$C$$
$$\{b \equiv (\forall i,j \in [0,n).\ (a[i] \wedge \neg a[j] \ \Rightarrow \ i < j))\}$$

dove a è un array di booleani costante, n è una costante naturale e b una variabile booleana.

In primo luogo è opportuno trovare una formulazione equivalente, ma più semplice della condizione:

$$\forall i,j \in [0,n).\ (a[i] \wedge \neg a[j] \ \Rightarrow \ i < j).$$

Abbiamo che:

$$\begin{array}{ll}
 & \forall i,j \in [0,n).\ (a[i] \wedge \neg a[j] \ \Rightarrow \ i < j) \\
\equiv & \quad \{ \text{ Contrapposizione } \} \\
 & \forall i,j \in [0,n).\ \neg(i < j) \ \Rightarrow \ \neg(a[i] \wedge \neg a[j]) \\
\equiv & \quad \{ \text{ Calcolo } \} \\
 & \forall i,j \in [0,n).\ i \geq j \ \Rightarrow \ \neg(a[i] \wedge \neg a[j]) \\
\equiv & \quad \{ \text{ Calcolo } \} \\
 & \forall i \in [0,n).\ \forall k \in [0,n-i).\ \neg(a[i] \wedge \neg a[i+k]) \\
\equiv & \quad \{ \text{ Induzione su } n \} \\
 & \forall i \in [0,n).\ \neg(a[i] \wedge \neg a[i+1])
\end{array}$$

L'ultimo passaggio, pur essendo semplice da comprendere, richiede una prova formale che procede per induzione su n (il lettore può, per esercizio, esplicitare la prova).

Il problema può dunque essere risolto con una ricerca lineare incerta di un indice $i \in [0, n-1)$ tale che $a[i] \wedge \neg a[i+1]$. Se la ricerca ha successo, la sequenza non soddisfa la proprietà in analisi e quindi b deve assumere valore falso. In caso contrario b deve assumere valore vero. Si ottiene così il programma:

```
{pre : x ∈ [0, n − 1).def(a[x] and nota[i + 1]))}
   x, trovato := 0, false;
{inv : Inv} {ft : n − 1 − (x + int(trovato))}
   while x < n − 1 and not trovato do
      if a[x] and not a[x + 1]
         then trovato := true
         else x := x + 1
      fi
   endw
{post}
   b := (x = n − 1)
{b ≡ (∀i, j ∈ [0, n). (a[i] ∧ ¬a[j]  ⇒  i < j))}
```

dove la precondizione *pre* è chiaramente soddisfatta e la postcondizione del ciclo, indicata con $G(x)$ la guardia $a[x]$ *and not* $a[x+1]$, è:

$$post\text{: } x \in [0, n-1] \ \wedge \ (\forall j \in [0,x).\ \neg G(j)) \ \wedge (x < n-1 \ \Rightarrow \ G(x))$$

e quindi, è immediato provare che:

$$post \ \Rightarrow \ ((x = n-1) \ \equiv \ (\forall j \in [0, n-1).\ \neg G(j)))$$

ovvero, in base alle considerazioni iniziali:

$$post \ \Rightarrow \ ((x = n-1) \ \equiv \ (\forall j \in [0, n-1).\ \neg(a[j] \wedge \neg a[j+1])))$$

Pertanto l'assegnamento $b := (x = n-1)$ completa il programma in modo da soddisfare la tripla data.