Dispense del corso di
Tecniche di Sicurezza basate sui Linguaggi

Daniele Gorla Ivano Salvo

17 aprile 2009

Sommario

Queste dispense sono una elaborazione di [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]. In particolare, il capitolo 1 è basato su [1], il capitolo 2 rielabora [2, 3, 4] (in particolare, [4] è stato adattato alle convenzioni e notazioni seguite nei capitoli successivi), il capitolo 3 segue fedelmente [5], il capitolo 4 segue fedelmente le prime 5 sezioni di [6], il capitolo 5 rielabora le prime 5 sezioni di [7], i capitoli 6 e 7 seguono fedelmente [8], mentre il capitolo 8 contiene cenni di [9, 10] e buona parte di [11].
Una prima stesura in latex è stata redatta nel corso dell'AA 2006/2007 dagli studenti Alessandro Cammarano, Andrea Cerone, Aleandro Morzilli e Bruno Vavalà, che ringraziamo per l'ottimo lavoro svolto.

Indice

Capitolo 1

Introduzione

Il termine *sicurezza* è uno dei più inflazionati in informatica, anche perchè viene spesso usato per raggruppare assieme problematiche piuttosto diverse. Tanto per citare le più classiche e diffuse:

Confidenzialità (o segretezza): i dati devono poter essere acceduti solamente dagli utenti autorizzati. In altri termini, nessun intruso può accedere a dati senza autorizzazione. Informazioni tipiche da proteggere sono dati sensibili quali una password, il saldo del proprio conto corrente, ...

Autenticità: garantire che un certo dato è stato prodotto da un certo utente. In altri termini, nessun intruso può fingersi quell'utente e produrre come tale dati falsificati. Tipico esempio di questa problematica è il distinguere un aggiornamento software da un virus.

Integrità: garanzia che un certo dato non è stato modificato. In altri termini, nessun intruso può modificare impropriamente dati non suoi. Ad esempio, evitare che un commerciante si accrediti più soldi di quelli che un cliente gli deve.

Disponibilità: garantire risorse a ogni utente che dispone le credenziali per richiederle. In altri termini, nessun intruso può "consumare una risorsa. Ad esempio, evitare denial-of-service (tipo riempire una mailbox con spam, sovraccaricare un server, ...).

Freschezza: rendere impossibile la duplicazione di informazioni. In altri termini, nessun intruso può replicare un dato. Ad esempio, evitare attacchi basati sulla ripetizione di un dato valido (ad esempio, il login).

Elemento comune in tutte queste problematiche è la presenza di un *intruso* che cerca di alterare il corretto funzionamento di un sistema.
Alcune problematiche possono essere risolte in maniera soddisfacente tramite tecniche crittografiche: ad esempio, l'autenticità può essere garantita tramite meccanismi di firma digitale, oppure l'integrità è ottenibile tramite meccanismi di cifratura a chiave pubblica/privata. Altre problematiche sono invece

tipicamente controllate a livello software, cioè dal sistema operativo. Un tempo, i sistemi operativi erano costituiti da un *kernel* relativamente semplice e di dimensioni ridotte. In questo modo, si teneva conto del seguente principio fondamentale:

> **Minimal Trusted Computing Base:** il grado di sicurezza di un meccanismo è inversamente proporzionale alle sue dimensioni.

Al contrario, i sistemi operativi attuali violano tale principio poichè hanno un codice sorgente molto vasto e spesso molto complicato. Inoltre, molti servizi non sono più implementati all'interno del kernel ma sono offerti da programmmi esterni. Il kernel non può dunque essere in grado di adottare meccanismi di sicurezza adeguati per la protezione di tutti i servizi. Infine, l'approccio tipico dei sistemi operativi consiste nell'assegnare a un utente tutti i privilegi che gli spettano: questo viola un altro fondamentale principio della sicurezza informatica:

> **Minimo Privilegio:** Durante l'esecuzione, ogni entità dovrebbe avere visibilità delle sole risorse e informazioni necessarie all'espletamento del compito corrente.

Assegnare sempre tutti i privilegi a un utente rende facile, ad esempio, la propagazione dei virus o il commettere gravi errori in maniera accidentale.
Un modo soddisfacente per risolvere tutti questi problemi consiste nell'affiancare al kernel una porzione affidabile di codice che monitorizzi l'esecuzione dei programmi. In questo modo, il principio del Minimal Trusted Computing Base sarebbe soddisfatto (la nuova porzione di codice è piccola) e si potrebbe facilmente implementare il principio del Minimo Privilegio, essendo questo approccio una tecnica dinamica. Il problema di questa soluzione è legato alla sua inefficienza: ogni azione monitorata richiede una system call spesso inutile, perchè la maggior parte delle azioni sono innocue dal punto di vista della sicurezza.
Per ovviare a questi inconvenienti senza rinunciare alla possibilità di avere tecniche solide per la sicurezza di un sistema, si sono sviluppate negli anni tecniche di analisi e riscrittura di programmi. L'insieme di tutti questi approcci va sotto il nome di *Tecniche di Sicurezza basate sui Linguaggi* o, in inglese, **Language Based Security**. In generale, si possono identificare tre tipi di approccio:

Analisi statica: Vengono effettuati dei controlli sul codice prima di mandarlo in esecuzione, in modo da garantire che un programma che ha superato la fase di analisi statica non genererà errori una volta mandato in esecuzione: questa tecnica è tipica dei compilatori, che effettuano ad esempio il controllo dei tipi. Tali approcci possono essere arbitrariamente potenti ed espressivi; il problema però è che queste caratteristiche sono inversamente proporzionali all'efficienza e alla decidibilità della tecnica stessa.

Riscrittura di programmi: I programmi vengono riscritti in maniera tale da poter essere resi sicuri. La riscrittura di programmi, tuttavia, provoca l'aggiunta di nuovi controlli all'applicazione che ne aumentano il costo

computazionale; non è però necessario porre controlli su ogni azione, ma solo su quelle più strettamente collegate a questioni di sicurezza. Uno svantaggio di questo approccio è il livello al quale viene applicato: lavorare sul codice oggetto è difficile, lavorare sul codice di alto livello richiede che il compilatore sia affidabile (problema, poichè il compilatore è solitamente un programma molto complesso).

Compilatori certificanti: compilatori che, assieme al codice, generano un certificato che ne dimostra la sicurezza. Si noti che con questo approccio nè il compilatore nè il generatore del certificato devono essere sicuri; al contrario, il verificatore del certificato deve essere sicuro, ma in generale è piuttosto piccolo.

In queste dispense ci concentreremo soprattutto sul primo approccio e parleremo brevemente del secondo; il terzo viene lasciato come approfondimento al lettore interessato.

Parte I

Flusso d'informazioni

Capitolo 2

Varie nozioni di Non interferenza

Concentriamoci sul problema della confidenzialità. A prima vista, potrebbe sembrare che questo problema sia facilmente risolvibile con meccanismi di cifratura a chiave pubblica/privata. In realtà, il problema di rendere segreto un dato cifrandolo sposta solo il problema sul rendere segreta la chiave. Inoltre, anche assumendo che la chiave sia mantenuta segreta, resta il problema dei *flussi di informazione* (o, in inglese, **information flow**). Si immagini ad esempio di avere due variabili, che per semplicità denoteremo con **P** e **S**, la prima pubblica e la seconda segreta, entrambe di tipo *bool*, e si consideri il seguente programma:

```
if (S) { P = true } else { P = false }
```

In questo codice, il segreto non è direttamente reso pubblico tramite una copia esplicita, tipo

```
P = S
```

ma la copia avviene in maniera indiretta, nascondendola tramite un flusso di codice complesso. Entrambi i codici descrivono un flusso di informazione; il secondo implementa un canale di informazione esplicito, il primo un canale implicito e in questo caso si parla di *cover channel*. I cover channel sono meccanismi che forniscono un flusso di informazioni tramite aspetti "secondari" di una computazione, quali il flusso di codice, aspetti legati alla terminazione o alla durata dell'esecuzioe di un programma, etc. Essi sono difficili da individuare anche con tecniche di verifica automatica avanzate; ad esempio i seguenti codici rappresentano dei cover channel abbastanza complessi:

```
if (S) { while(true) }
```

```
if (S) { wait(1000) }
```

Il primo frammento di codice rappresenta un *termination cover channel*: un utente che esegue il programma può accertare il valore della variabile S semplicemente osservando la terminazione del programma. Il secondo frammento di codice denota invece un *timing cover channel*: in base al tempo d'esecuzione del programma, si riescono a ottenere informazioni sul segreto.
Come sempre avviene in sicurezza, bisogna fare delle assunzioni sul potere dell'attaccante: un attaccante di potere illimitato vanificherebbe ogni tentativo di protezione di un qualsiasi sistema. Una volta fissato ciò che un attaccante può osservare, si può procedere a stabilire tecniche in grado di svelare buchi nella sicurezza di un programma, rispetto al tipo di attacchi assunti. Dovrebbe essere infatti chiaro che, ad esempio, i termination/timing cover channels mostrati precedentemente non mettono a rischio la sicurezza di un sistema se l'attaccante può solo leggere il valore finale delle variabili pubbliche di un programma.
Un altro aspetto fondamentale da fissare è l'ambiente di programmazione in cui si lavora: caratteristiche avanzate, tipo primitive per la programmazione concorrente o probabilistiche, possono permettere di montare nuovi attacchi. Pertanto, partiremo dal modello computazionale più elementare (un semplice ambiente di programmazione imperativa) e in esso definiremo varie nozioni di non interferenza, in base al potere osservazionale dell'attaccante. In seguito considereremo anche aspetti più avanzati e l'impatto che hanno sulle problematiche di non interferenza.

2.1 Programmazione Imperativa

Un programma imperativo ha lo scopo di modificare una memoria di input, trasformandola in una memoria di output. Ad alto livello, la *memoria* può essere vista come un'associazione di *valori* a *variabili*.

Definizione 2.1.1.

Sia $\mathcal{V}$ l'insieme delle variabili e sia Val l'insieme dei valori assegnabili ad una variabile. Una memoria μ è una funzione

$$\mu : \mathcal{V} \to Val$$

che associa ad ogni variabile il suo valore.

Per semplicità, nei programmi che analizzeremo le variabili saranno di due tipi: segrete e pubbliche. Spesso, chiameremo le prime *variabili alte* e le seconde *variabili basse* (nel senso che il loro livello di segretezza è alto o basso, rispettivamente).

Definizione 2.1.2.

Dato l'insieme $\mathcal{V}$ delle variabili, il livello di una variabile è un elemento dell'insieme $\{H, L\}$. Un tipaggio delle variabili è una funzione $\mathcal{L} : \mathcal{V} \to \{H, L\}$ che associa ad ogni variabile un livello.

In seguito, per semplificare la lettura, indicheremo con **H** una variabile di tipo alto e con **L** una di tipo basso. L'insieme delle variabili $\mathcal{V}$ è bipartizionabile in due sottoinsiemi:

- $\mathcal{V}_{low} = \{v \in \mathcal{V} \mid \mathcal{L}(v) = L\}$
- $\mathcal{V}_{high} = \{v \in \mathcal{V} \mid \mathcal{L}(v) = H\}$

In astratto un programma imperativo è una funzione che trasforma memorie. Cioè, data una memoria μ iniziale, un programma imperativo o termina in uno stato di memoria μ', oppure non termina.

Definizione 2.1.3.

Un programma imperativo è una funzione parziale che associa ad una memoria di input una memoria di output:

$$\Pi : \mathcal{M} \rightharpoonup \mathcal{M}$$

dove $\mathcal{M}$ denota l'insieme delle memorie.

2.2 Non interferenza

Il bipartizionamento dell'insieme delle variabili in alte e basse consente di analizzare il grado di sicurezza di un programma. Infatti, possiamo affermare che un programma è sicuro quando, cambiando i valori delle variabili di livello alto, non vengono influenzate le variabili di livello basso. Cerchiamo ora di formalizzare opportunamente questa idea.

Definizione 2.2.1.

Low Equivalenza: *Due memorie μ e μ' sono Low Equivalenti, scritto $\mu =_L \mu'$, se, presa comunque una variabile v di livello **low**, risulta $\mu(v) = \mu(v')$. In formula:*

$$\mu =_L \mu' \Leftrightarrow \forall v \in \mathcal{V}_{low}\ \mu(v) = \mu'(v)$$

Supponiamo di avere un programma imperativo Π e che tale programma venga eseguito a partire da due memorie μ e μ' tra loro low equivalenti. Denotiamo le rispettive memorie al termine della computazione con υ e υ'. Se le memorie μ' e υ' non sono tra di loro low equivalenti, allora Π non rispetta il concetto di sicurezza illustrato precedentemente, in quanto i valori delle variabili basse sono stati influenzati da un qualche sottoinsieme non vuoto delle variabili alte: in tal caso diremo che il programma Π è *interferente*.
Come abbiamo già detto, ogni covert channel che si vuole eliminare richiede un'opportuna definizione di non-interferenza; partiamo dai flussi di informazione provenienti da implicit flows.

Definizione 2.2.2.

Non Interferenza, per flussi impliciti: *Un programma* Π *è non interferente se e solo se* $\forall \mu, \upsilon : (\Pi(\mu) = \mu', \Pi(\upsilon) = \upsilon',\ \mu =_L \upsilon$ *si ha che* $\mu' =_L \upsilon'$.

Mostriamo il concetto di interferenza con un esempio: il programma Π definito come segue

```
P = false;
if (S) { P = true }
```

risulta essere interferente, se si assume il tipaggio $\mathcal{L}(P) = L$ e $\mathcal{L}(S) = H$. Siano infatti μ e υ due memorie tra loro low equivalenti; siano inoltre μ' e υ' le rispettive memorie ottenute al termine dell'esecuzione del programma Π. Possiamo distinguere tre casi:

1. $\mu(S) = \upsilon(S) = false$: allora $\mu'(P) = \upsilon'(P) = false$
2. $\mu(S) = \upsilon(S) = true$: allora $\mu'(P) = \upsilon'(P) = true$
3. $\mu(S) = true, \upsilon(S) = false$: allora $\mu'(P) = true,\ \upsilon'(P) = false$

Il terzo caso viola chiaramente la Definizione 2.2.2 e quindi il programma è interferente: partendo da memorie low equivalenti, le memorie finali non sono low equivalenti. E' stato quindi rilevato un cover channel dovuto a un flusso implicito.
Come è ovvio, covert channels dovuti ad altri aspetti non vengono catturati da questa definizione. Per esempio, si consideri il seguente termination cover channel:

```
P = true;
while(S) { };
P = false
```

Utilizzando il tipaggio dell'esempio precedente e analizzando il caso in cui $\mu(S) = true$, si ottiene che il programma non termina. Tuttavia, il programma risulta essere non interferente secondo la definizione (2.2.2), in quanto l'ipotesi (non soddisfatta) richiede che il programma termini. Per scoprire termination cover channels, occorre pertanto migliorare la definizione di non interferenza, e utilizzare la seguente:

Definizione 2.2.3.

Non Interferenza, sensibile alla terminazione: *Un programma* Π *è non interferente se, e solo se* $\forall \mu, \upsilon : \mu =_L \upsilon$, $\Pi(\mu) = \mu'$, *si ha che* $\mu' =_L \upsilon'$, *con* $\Pi(\upsilon) = \upsilon'$.

La nuova definizione di non interferenza permette di rilevare flussi basati anche su termination cover channel, come quello presentato nell'esempio precedente: infatti, essa richiede che alterazioni della parte alta di memorie iniziali low-equivalenti (i) non abbiano ripercussioni sulla parte basse delle memorie finali e (ii) non alterino la terminazione del programma.
Immaginiamo ora che l'attaccante possa fare osservazioni, oltre che sulla terminazione di una programma, anche sulle modalità di terminazione (cioè, abnorme o normale). Assumiamo inoltre che i booleani siano codificati con numeri interi (0 è il $false$, un qualsiasi numero diverso da 0 è il $true$). In tal caso, il seguente codice

```
P = 0;
if (1/S) { } { };
P = 1
```

sarebbe sicuro secondo la Definizione 2.2.3 ma conterrebbe comunque un flusso di informazione. Per ovviare a ciò, bisogna modificare la Definizione 2.2.3 richiedendo che $\Pi(\mu)$ e $\Pi(\upsilon)$ abbiano la stessa modalità di terminazione.
Se l'attaccante è anche in grado di osservare il tempo d'esecuzione di un programma, bisogna modificare le definizioni precedenti di non interferenza in modo da garantire che il tempo necessario per arrivare alla memoria finale sia lo stesso partendo da qualunque coppia di memorie low-equivalenti. Invece, se l'attaccante è in grado di eseguire il programma più volte e nel programma c'è qualche aspetto non-deterministico o probabilistico (tipo, generazione di numeri pseudo-casuali), bisogna modificare le definizioni di non interferenza in modo da garantire che la distribuzione di probabilità sulle memorie finali sia la stessa partendo da qualunque coppia di memorie low-equivalenti. In definitiva, se l'attaccante ha capacità di osservazione più raffinate (quindi, se si affrontano diversi tipi di cover channels), sarà necessario raffinare il concetto di non interferenza; questo poichè l'attaccante può evincere più informazioni osservando l'esecuzione di un

programma o, simmetricamente, un programma può (volontariamente o involontariamente) far filtrare più informazioni riguardo i suoi segreti e l'attaccante è in grado di cogliere tali informazioni.

2.3 Non interferenza e Integrità

Le problematiche relative al flusso delle informazioni sono tipicamente collegate al problema della segretezza: si vuole impedire che informazioni segrete vengano in qualche modo rese pubbliche. Tutti i idscorsi fatti finora e che faremo in seguito si adattano però molto bene anche a problematiche di *integrità*. In quest'ambito, non si vuole che valori di variabili corrotte influenzino variabili che devono restare integre. Questo ricorda molto le varie nozioni di non-interferenza discusse finora, dove valori di variabili alte non dovevano influenzare variabili basse. Ad esempio, assumendo che C denoti una variabile corrotta e I una variabile integra, un codice tipo

```
if (C) { I = true } else { I = false }
```

copia (in maniera implicita) il valore di una variabile corrotta in una variabile integra, e quindi viola la proprietà di integrità che vorremmo garantire.
Tale programma può essere definito insicuro usando una definizione formalmente identica a Definizione 2.2.2. Modificando il potere dell'osservatore, si renderanno com'è ovvio necessarie le corrispondenti modifiche di tale definizione. Si noti solo che, mentre nel caso della segretezza 'alto' voleva dire 'sicuro', nel caso dell'integrità 'alto'/'basso' vuol dire con un alto/basso rischio di corruttibilità.

Capitolo 3

Non interferenza in un linguaggio imperativo

3.1 Sintassi e semantica

Negli esempi precedenti si è più volte utilizzato del codice esplicito: tuttavia, non è stata definita ancora la sintassi del linguaggio che verrà utilizzato per l'analisi del flusso di informazioni. Per non appesantire troppo le notazioni, verrà introdotto un linguaggio minimale, garantendo comunque la *Turing Completezza* dello stesso.
Il linguaggio che andremo a presentare sarà composto di due componenti fondamentali: le espressioni e i comandi. Mentre le espressioni verranno utilizzate principalmente per effettuare calcoli e controlli, i comandi avranno lo scopo di definire il comportamento del programma; formalmente si hanno le seguenti definizioni.

Definizione 3.1.1.

Espressioni: *Un'espressione è definita dalla seguente grammatica:*

$$e \ ::= \ n \mid x \mid e_1 + e_2 \mid e_1 == e_2 \mid \cdots$$

dove $x \in \mathcal{V}, n \in Val$. *Denotiamo con* $\mathcal{E}$ *l'insieme delle espressioni così definito.*

Definizione 3.1.2.

Comandi: *Un comando è definito dalla seguente grammatica:*

$$c \ ::= \ x = e \mid c_1 \ ; \ c_2 \mid if \ (\ e \ ) \ \{ \ c_1 \ \} \ else \{ \ c_2 \ \} \mid while \ (\ e \ ) \{ \ c \ \}$$

dove $x \in \mathcal{V}, e \in \mathcal{E}$.

Un'espressione può essere una costante, una variabile, la somma di due espressioni, il confronto di due espressioni tramite l'uguaglianza, e così via; per semplicità sono state omesse altre possibili operazioni applicabili a due espressioni. I comandi usano le espressioni per assegnare un valore a una variabile (ad esempio il comando $x = x + 5$), oppure per controllare il flusso di esecuzione (ad esempio in $if \ (x == 5) \ \{\cdots\}\{\cdots\}$). Chiaramente, comandi semplici possono essere composti per dar vita ad un comando complesso tramite concatenazione di due comandi (ad esempio $x = x + 5; \ if(x == 5) \ \{\cdots\}$).
La semantica operazionale descrive l'evoluzione di un comando in una memoria; pertanto, è una relazione tra coppie (μ, c), dove μ è una memoria e c è un comando. Per far ciò, abbiamo dapprima bisogno di valutare espressioni a partire da una memoria, dal momento che in esse possono apparire dei riferimenti a delle variabili. Il risultato di tale valutazione apparterrà all'insieme dei valori Val, introdotto precedentemente.
Per indicare che un'espressione e valutata in una memoria μ dà il valore $n \in Val$, useremo la notazione

$$(\mu, e) \to n$$

E' possibile definire delle regole di inferenza per la valutazione delle espressioni per induzione sulla loro struttura, definita in (3.1.1).

Regola di inferenza 3.1.1.

$$\frac{}{(\mu, n) \to n} \qquad \frac{\mu(x) = n}{(\mu, x) \to n} \qquad \frac{(\mu, e_1) \to n_1 \quad (\mu, e_2) \to n_2}{(\mu, e_1 + e_2) \to n_1 + n_2}$$

$$\frac{(\mu, e_1) \to n \quad (\mu, e_2) \to n}{(\mu, e_1 == e_2) \to 1} \qquad \frac{(\mu, e_1) \to n_1 \quad (\mu, e_2) \to n_2}{(\mu, e_1 == e_2) \to 0} \ n_1 \neq n_2$$

I casi base sono quelli per cui un'espressione è una costante o una variabile: nel primo caso l'espressione ha come valore il valore della costante stessa; nel secondo caso l'espressione sarà valutata con il valore assegnato dalla memoria alla variabile stessa. Le altre regole di inferenza vengono definite a partire da queste due: ad esempio, se si vuole valutare un'espressione del tipo $e_1 + e_2$, si

dovrà dapprima valutare le singole espressioni, e poi procedere alla somma dei valori ottenuti. Per i confronti sono necessarie due regole di inferenza: se per esempio si considera il confronto di due espressioni tramite l'operatore ==, si ha che tale espressione andrà valutata con il valore 1 qualora le due espressioni vengano valutate con lo stesso valore, 0 altrimenti. Lasciamo stabilire al lettore le regole di inferenza per gli operatori omessi.
Passiamo ora ai comandi. A livello intuitivo, diremo che una coppia (μ, c) è in relazione con una memoria μ' se, dopo aver eseguito il comando c trovandosi nello stato di memoria μ, un programma termina nello stato di memoria μ'. Diremo invece che una coppia (μ, c) è in relazione con una coppia (μ', c') se, dopo aver eseguito il comando c trovandosi nella memoria μ, si finisce nella memoria μ' e si deve proseguire eseguendo il comando c'. In simboli:

- $(\mu, c) \to \mu'$
- $(\mu, c) \to (\mu', c')$

Regola di inferenza 3.1.2.

$$\frac{(\mu, e) \to n}{(\mu, x = e) \to \mu[x \leftarrow n]}$$

$$\frac{(\mu, c_1) \to \mu'}{(\mu, c_1; c_2) \to (\mu', c_2)} \qquad \frac{(\mu, c_1) \to (\mu', c_1')}{(\mu, c_1; c_2) \to (\mu', c_1'; c_2)}$$

$$\frac{(\mu, e) \to 1}{(\mu, if(e)\ \{c_1\}\ else\ \{c_2\}) \to (\mu, c_1)} \qquad \frac{(\mu, e) \to 0}{(\mu, if(e)\ \{c_1\}\ else\ \{c_2\}) \to (\mu, c_2)}$$

$$\frac{(\mu, e) \to 0}{(\mu,\ while(e)\ \{c\}) \to \mu} \qquad \frac{(\mu, e) \to 1}{(\mu,\ while(e)\ \{c\}) \to (\mu,\ c; while(e)\ \{c\})}$$

Anche in questo caso la definizione delle regole di inferenza avviene per induzione. L'unico caso base è quello in cui il comando eseguito è l'assegnamento di una variabile: se la coppia *(memoria, comando)* presa in considerazione è $(\mu, x = e)$, allora si dovrà valutare l'espressione e, e si transiterà in una nuova memoria che differisce da μ per il solo valore della variabile x: per semplicità di notazione, indicheremo tale memoria come $\mu[x \leftarrow n]$, dove n è il risultato della valutazione dell'espressione.
Per quanto riguarda la concatenazione di comandi, supponiamo di avere una concatenazione del tipo $c_1; c_2$. Limitiamo per il momento la discussione all'esecuzione del solo comando c_1: questi potrà terminare in un solo passo, portando la memoria μ alla memoria μ'; differentemente c_1 potrebbe richiedere più passi

per terminare, per cui l'esecuzione di $c1$ nello stato di memoria μ porterebbe a dover eseguire la parte rimanente del comando, che denoteremo con c_1', in una nuova configurazione di memoria μ'. Tornando al comando composto $c_1; c_2$, nel primo caso si dovrà procedere a eseguire c_2 nello stato μ', mentre nel secondo caso si dovrà procedere eseguendo il comando $c_1'; c_2$ nella memoria μ'.
Il controllo del flusso di esecuzione tramite un'istruzione di tipo if richiede anch'esso due regole di inferenza: supponiamo di partire in uno stato di memoria μ e di eseguire il codice $if(e)\ \{c_1\}\ else\ \{c_2\}$: se la valutazione dell'espressione e è pari al valore 0, allora si dovrà procedere eseguendo il codice c_1, in caso contrario si dovrà eseguire il codice c_2; in ogni caso la memoria sulla quale eseguire il codice successivo all'istruzione di if sarà sempre μ
Infine, un ciclo $while$ è gestita tramite la concatenazione di comandi: ancora una volta, partendo da una coppia del tipo $(\mu,\ while(e)\ \{c\})$, si dovrà valutare l'espressione e nella memoria μ, quindi dovremo aspettarci nuovamente due regole di inferenza. Se tale espressione viene valutata 0, allora ci si aspetta di dover uscire dal ciclo di iterazione e di restare nella memoria μ. Tuttavia, se tale espressione viene valutata 1, allora si dovrà eseguire la concatenazione di comandi c; $while(e)\ \{c\}$ nello stato di memoria μ iniziale. A questo punto si dovrà valutare una concatenazione di comandi, caso per il quale sono già state fornite le regole di inferenza

E' talvolta necessario considerare più passi operazionali alla volta; a tal fine, usiamo la chiusura transitiva e riflessiva della relazione di transizione $\rightarrow$, denotata con $\rightarrow^*$. Essa può essere definita tramite delle ovvie regole di inferenza.

Regola di inferenza 3.1.3.

$$\frac{(\mu, c_1) \rightarrow (\mu', c_2)}{(\mu, c_1) \rightarrow^* (\mu', c_2)} \qquad \frac{(\mu, c_1) \rightarrow \mu'}{(\mu, c_1) \rightarrow^* \mu'}$$

$$\frac{(\mu, c_1) \rightarrow^* (\mu', c_2) \quad (\mu', c_2) \rightarrow^* (\mu'', c_3)}{(\mu, c_1) \rightarrow^* (\mu'', c_3)}$$

$$\frac{(\mu, c_1) \rightarrow^* (\mu', c_1) \quad (\mu', c_1) \rightarrow^* \mu''}{(\mu, c_1) \rightarrow^* \mu''}$$

3.2 Semantica operazionale: esempi

Esaminiamo ora degli esempi, con lo scopo di rendere più chiaro l'utilizzo delle regole di inferenza e mostrare come si costruisce un albero di derivazione per un

passo computazionale di un dato programma. Come primo esempio mostreremo, data un'espressione e uno stato di memoria, come sia possibile trarne una valutazione utilizzando le regole di inferenza definite nel paragrafo precedente. Sia dato lo stato di memoria μ, tale che:

- $\mu(x) = 5$
- $\mu(y) = 3$

dove $x, y \in Var$. In tale memoria vogliamo valutare l'espressione

$$x + 3 == y + 5$$

La regola di inferenza per il test di uguaglianza richiede che, per valutare tale espressione, è necessario dapprima valutare le singole espressioni che vengono poste a confronto. I possibili casi sono due: o la valutazione di $x + 3$ darà un valore uguale alla valutazione dell'espressione $y + 5$ e l'intera espressione verrà valutata 1, oppure verrà valutata 0.
Passiamo allora a valutare l'espressione $x+3$: la regola di inferenza per la somma stabilisce che bisogna valutare singolarmente le due espressioni che compongono la somma. Pertanto, abbiamo

$$\dfrac{\dfrac{\mu(x) = 5}{(\mu, x) \to 5} \qquad \dfrac{}{(\mu, 3) \to 3}}{(\mu, x + 3) \to 8}\ 8{=}5{+}3$$

Un albero di derivazione simile può essere costruito, in maniera analoga, per l'espressione $y + 5$: l'albero risultante sarà il seguente

$$\dfrac{\dfrac{\mu(y) = 3}{(\mu, y) \to 3} \qquad \dfrac{}{(\mu, 5) \to 5}}{(\mu, y + 5) \to 8}\ 8{=}3{+}5$$

Infine, le valutazioni di tali espressioni dovranno essere confrontate. Poiché effettivamente la valutazione di entrambe le espressioni è 8, allora si avrà che l'intera espressione verrà valutata con 1.

$$\dfrac{\dfrac{\dfrac{\mu(x) = 5}{(\mu, x) \to 5} \qquad \dfrac{}{(\mu, 3) \to 3}}{(\mu, x + 3) \to 8} \qquad \dfrac{\dfrac{\mu(y) = 3}{(\mu, y) \to 3} \qquad \dfrac{}{(\mu, 5) \to 5}}{(\mu, y + 5) \to 8}}{(\mu, x + 3 == y + 5) \to 1}\ 8{=}8$$

Mostriamo ora il comportamento di un comando a partire da una memoria iniziale μ. Per denotare una memoria μ useremo la la scrittura

$$[v_1 \leftarrow n_1, v_2 \leftarrow n_2, \cdots] \qquad \text{per } v_i \in \mathcal{V}, n_i \in Val$$

Inoltre, indicheremo con $\mu[x \leftarrow n]$ la memoria μ' tale che $\mu')(x) = n$ e $\mu'(y) = \mu(y)$, per ogni $y \neq x$.
Analizziamo adesso il comportamento di un programma utilizzando come insieme di valori $Val = \mathbb{Z}$. Il programma per il quale costruiremo l'albero di derivazione è il seguente:

```
P = 0;
while (P < S) { P = P + 1 }
```

Ovviamente questo programma copia, tramite un cover channel, il valore della variabile **S** in **P**. Prendiamo come memoria di partenza $\mu \stackrel{\triangle}{=} [P \leftarrow 1, S \leftarrow 1]$.
Il programma è composto dalla concatenazione di due comandi, $P = 0$ e un ciclo d'iterazione. Secondo la regola di inferenza per la concatenazione si dovrà prima procedere con l'analisi del comando $P = 0$ e utilizzare la memoria risultante dall'esecuzione di questi per costruire la derivazione del ciclo d'iterazione.
Per l'istruzione $P = 0$ è semplice costruire l'albero di derivazione; la regola di inferenza per l'assegnamento ci dice che, partendo da una memoria μ ed effettuando un assegnamento sulla variabile **P**, si dovrà valutare l'espressione alla destra dell'assegnamento (ottenendo il valore n) e si terminerà nella memoria $\mu[P \leftarrow n]$. In questo caso si ottiene:

$$\frac{(\mu, 0) \rightarrow 0}{(\mu, P = 0) \rightarrow \mu[P \leftarrow 0]}$$

La derivazione dell'albero corrispondente al comando $while(P < S)\ \{P = P{+}1\}$ dovrà pertanto essere effettuata partendo dalla memoria $\mu' = \mu[P \leftarrow 0]$. In tal caso verrà valutata l'espressione $P < S$: ovviamente risulterà che il risultato di tale derivazione è 1 e dunque, secondo la regola di inferenza per il while, si deve procedere a valutare l'esecuzione del comando $P = P + 1; while(P < S)\{\cdots\}$ nella configurazione di memoria μ':

$$\frac{\dfrac{\mu'(P) = 0}{(\mu', P) \rightarrow 0} \qquad \dfrac{\mu'(S) = 1}{(\mu', S) \rightarrow 1}}{(\mu', P < S) \rightarrow 1}$$

$$\overline{(\mu',\ while(P < S)\ \{P = P + 1\}) \rightarrow (\mu',\ P = P + 1\ ;\ while(P < S)\ \{P = P + 1\})}$$

A questo punto dovremo calcolare la memoria risultante dall'esecuzione del comando $P = P + 1\ ;\ while(P < S)\ \{P = P + 1\}$ a partire da μ'. Dobbiamo dapprima valutare l'esecuzione dell'istruzione $P = P + 1$, in modo da determinare la memoria sulla quale andrà eseguito nuovamente il ciclo di iterazione. L'albero di derivazione che si ottiene è il seguente:

$$\frac{\dfrac{\dfrac{\mu'(P) = 0}{(\mu', P) \rightarrow 0} \qquad \dfrac{}{(\mu', 1) \rightarrow 1}}{(\mu', P + 1) \rightarrow 1}}{(\mu', P = P + 1) \rightarrow \mu'[P \leftarrow 1]}$$

Possiamo ora procedere a valutare nuovamente il ciclo while sulla memoria $\mu'' \triangleq [P \leftarrow 1, S \leftarrow 1]$. Bisognerà nuovamente costurire un albero di derivazione per il while, questa volta utilizzando la memoria di partenza μ''. In questo caso, si ottiene che la valutazione dell'espressione $P < S$ è pari a 0 e l'albero di derivazione risultante è il seguente:

$$\dfrac{\dfrac{\dfrac{\mu''(P) = 1}{(\mu'', P) \to 1} \quad \dfrac{\mu''(S) = 1}{(\mu'', S) \to 1}}{(\mu'', P < S) \to 0}}{(\mu'',\ while(P < S)\ \{P = P + 1\}) \to \mu''}$$

L'ultimo giudizio inferito mostra che, una volta effettuata la transizione, si termina in una configurazione di memoria μ'' senza alcun comando da eseguire. Dunque lo stato μ'' rappresenta la memoria di output del programma analizzato. Volendo riepilogare le transizioni effettuate durante l'esecuzione dell'intero programma si ottiene allora:

$$\begin{array}{ll} (\mu, P = 0;\ while\ (\ P < S\ )\ \{\ P = P + 1\ \}\ ) & \to \\ (\mu', while\ (\ P < S\ )\ \{\ P = P + 1\ \}\ ) & \to \\ (\mu', P = P + 1; while\ (\ P < S\ )\ \{\ P = P + 1\ \}\ ) & \to \\ (\mu'', while\ (\ P < S\ )\ \{\ P = P + 1\ \}\ ) & \to \mu'' \end{array}$$

Viene così mostrato che il valore della variabile **S**, pari a 1, viene effettivamente copiato nella variabile **P**. Il lettore provi a generalizzare il risultato e mostrare che, indipendentemente dalla memoria di partenza e assumendo $S \geq 0$ in ogni caso, il programma copia il valore della variabile **S** in **P**.[1]

3.3 Controllo dei tipi

Abbiamo introdotto la nozione di non interferenza come garanzia di assenza di flussi impliciti in programmi imperativi. Adesso si vuole studiare un metodo per scoprire se un programma è interferente, in modo da non permetterne l'esecuzione. Per semplicità, continueremo a considerare due soli livelli di variabili, corrispondenti agli elementi dell'insieme $\{H, L\}$; tuttavia il discorso è generalizzabile al caso di un reticolo con un numero arbitrario di livelli, dove l'ordinamento sugli elementi del reticolo è denotato con $\prec$. Nel nostro caso particolare avremo che $L \prec H$.
Il nostro obiettivo è quello di poter esaminare il codice di un programma e determinare se il programma è interferente o meno. Ciò verrà fatto tramite un sistema di tipi; in particolare, si vuole stabilire, dato un assegnamento di livelli di sicurezza alle variabili γ, se è possibile assegnare un tipo a una espressione o a un comando, e in caso affermativo qual è questo tipo.
Alle espressioni si assegnerà un tipo elementare τ (per $\tau \in \{L, H\}$), corrispondente ad uno dei livelli di sicurezza assunto per le variabili (nel nostro caso

[1] Suggerimento: si proceda per induzione sul valore di **S**.

solo L e H); l'idea è che una espressione τ contiene solo variabili di livello τ e costanti. Ai comandi si assegnerà un tipo τcmd, con l'idea che in un comando τcmd si assegnano valori di livello τ solo variabili di livello τ.
Possiamo adesso discutere le regole di inferenza che permettono di assegnare un tipo ad un'espressione; successivamente si esaminerà il caso dei comandi. Per indicare che, dato un tipaggio γ ad un'espressione **e** viene assegnato il tipo τ, utilizzeremo la notazione

$$\gamma \vdash e : \tau$$

Regola di inferenza 3.3.1.

$$\frac{}{\gamma \vdash n : \tau} \qquad \frac{\gamma(x) = \tau}{\gamma \vdash x : \tau} \qquad \frac{\gamma \vdash e_1 : \tau \quad \gamma \vdash e_2 : \tau}{\gamma \vdash e_1 \theta e_2 : \tau}$$

Il caso più semplice da esaminare è quello in cui l'espressione è un semplice valore: ad una costante può essere assegnato un qualsiasi valore $\tau \in \{H, L\}$. Nel caso in cui invece l'espressione è una variabile si dovrà semplicemente osservare il livello della variabile, $\gamma(x)$. Per concludere, abbiamo un'unica regola per ogni operatore binario (sia esso logico che aritmertico): dato un qualsiasi operatore binario θ e due espressioni e_1 ed e_2 di tipo τ, si ha che $e_1 \theta e_2$ ha tipo τ.

Passiamo ora al tipaggio dei comandi. Le regole di inferenza sono le seguenti:

Regola di inferenza 3.3.2.

$$\frac{\gamma \vdash x : \tau \quad \gamma \vdash e : \tau}{\gamma \vdash x = e : \tau cmd} \qquad \frac{\gamma \vdash c_1 : \tau cmd \quad \gamma \vdash c_2 : \tau cmd}{\gamma \vdash c_1; c_2 : \tau cmd}$$

$$\frac{\gamma \vdash e : \tau \quad \gamma \vdash c_1 : \tau cmd \quad \gamma \vdash c_2 : \tau cmd}{\gamma \vdash if(e)\ \{c_1\}\ else\ \{c_2\} : \tau cmd} \qquad \frac{\gamma \vdash e : \tau \quad \gamma \vdash c : \tau cmd}{\gamma \vdash while(e)\ \{c\} : \tau cmd}$$

Un assegnamento $x = e$ è lecito solo se si assegnano valori τ a variabili τ; il risultato sarà un comando τcmd. La concatenazione dei comandi può essere definita in maniera simile: se si hanno due comandi c_1 e c_2 entrambi di tipo τcmd, allora il comando $c_1; c_2$ sarà ancora un comando di tipo τcmd. Se si ha invece il comando $if(e)\ \{c_1\}\ else\ \{c_2\}$, allora sarà necessario effettuare un controllo anche sul tipo dell'espressione e: si otterrà che l'intero comando è di tipo τcmd se, e solamente se, l'espressione e è di tipo τ, e i comandi c_1 e c_2 sono entrambi di tipo τcmd; il caso del *while* è analogo.

Definizione 3.3.1.

Un comando c si dice ben tipato rispetto a un tipaggio γ (o, analogamente, che c è tipabile in γ) se $\gamma \vdash c : \tau cmd$, per qualche $\tau \in \{L, H\}$. Un comando c è tipabile se esiste un tipaggio γ rispetto a cui c è ben tipato.

Vale a dire, un programma si dice ben tipato quando è possibile assegnarvi un tipo secondo le regole di inferenza appena discusse.

3.4 Tipabilità: esempi

Sarà utile anche in questo caso analizzare alcuni esempi, in modo da poter mostrare la relazione che intercorre tra comandi ben tipati e comandi non interferenti. Consideriamo dapprima l'assegnamento

```
P = S + 1
```

e supponiamo $\gamma(P) = L, \gamma(S) = H$. Tale assegnamento va rigettato, in quanto il comando risulta interferente (il valore di una variabile alta è copiato in una variabile bassa). Se andiamo a costruire gli alberi di derivazione per **P** e per **S + 1**, otteniamo

$$\gamma \vdash P : L \qquad \gamma \vdash S + 1 : H$$

Notiamo infatti che mentre la variabile da assegnare è di tipo **L**, all'espressione viene assegnato il tipo **H**. Questo non rende applicabile la regola di tipaggio per l'assegnamento (l'unica applicabile qui, per via della struttura sintattica del comando) e il comando risulta non tipabile, come desiderato.
Come secondo esempio, consideriamo nuovamente il comando (interferente)

```
P = 0;
while (P < S) {
        P = P + 1
}
```

e vediamo se esso è tipabile rispetto al tipaggio $\gamma(P) = L, \gamma(S) = H$. Analizziamo dapprima il comando $P = 0$: ad esso è assegnabile solamente il tipo **Lcmd**. Infatti

$$\frac{\dfrac{\gamma(P) = L}{\gamma \vdash P : L} \qquad \gamma \vdash 0 : L}{\gamma \vdash P = 0 : Lcmd}$$

Dunque, affinché l'intero comando risulti ben tipato, si dovrà poter assegnare anche al while il tipo **Lcmd**. Ciò richiede le seguenti premesse:

- $\gamma \vdash P < S : L$
- $\gamma \vdash P = P + 1 : Lcmd$

Mentre per il secondo caso è possibile costruire un albero di derivazione che porti a $\gamma \vdash P = P + 1 : Lcmd$ (il lettore provi a effettuarne la derivazione), non è possibile trovarne uno per il confronto $P < S$. Si hanno infatti i seguenti alberi:

$$\frac{\gamma(P) = L}{\gamma \vdash P : L} \qquad \frac{\gamma(S) = H}{\gamma \vdash S : H}$$

Come speravamo, il programma (che è interferente) risulta non tipabile. In realtà questo è un risultato generale che verrà provato più avanti: nessun programma interferente è *tipabile*.

Tuttavia, il sistema di tipi che abbiamo presentato è troppo restrittivo per i nostri scopi, in quanto è possibile trovare una vasta quantità di programmi non interferenti che risultano non tipabili. Si consideri ad esempio il comando:

```
P = 0;
while (S1 < S2) {
        S1 = S1 + 1
}
```

e si prenda il tipaggio $\gamma(P) = L, \gamma(S1) = \gamma(S2) = H$. Tale programma rispetta la definzione di non interferenza; tuttavia non è tipabile, in quanto non è possibile dare all'assegnamento $P = 0$ e al ciclo d'iterazione lo stesso tipo. Proviamo infatti a costruire l'albero di inferenza: per l'assegnamento il lavoro è già stato svolto e si ha $\gamma \vdash P = 0 : Lcmd$. Per il while, si dovranno ancora una volta derivare i tipi dell'espressione e del comando iterato. Si ha allora:

$$\frac{\dfrac{\gamma(S1) = H}{\gamma \vdash S1 : H} \qquad \dfrac{\gamma(S2) = H}{\gamma \vdash S2 : H}}{\gamma \vdash S1 < S2 : H}$$

e

$$\frac{\dfrac{\gamma(S1) = H}{\gamma \vdash S1 : H} \qquad \dfrac{\dfrac{\gamma(S2) = H}{\gamma \vdash S2 : H} \qquad \dfrac{}{\gamma \vdash 1 : H}}{\gamma \vdash S1 + 1 : H}}{\gamma \vdash S1 = S1 + 1 : Hcmd}$$

Pertanto si ha che il ciclo d'iterazione è ben tipato, poiché

$$\frac{\gamma \vdash S1 < S2 : H \quad \gamma \vdash S1 = S1 + 1 : Hcmd}{\gamma \vdash while(S1 < S2)\ \{S1 = S1 + 1\} : Hcmd}$$

Si ha però che l'intero comando non è tipabile: da un lato abbiamo infatti $\gamma \vdash P = 0 : Lcmd$, mentre dall'altro si ha $\gamma \vdash while \cdots : Hcmd$, mentre si richiede che ai due comandi sia assegnato lo stesso tipo. Sarà necessario allora raffinare il sistema dei tipi che è stato presentato, in modo da consentire a comandi quali il precedente di risultare tipabili.

3.5 Raffinamento del sistema dei tipi

Come abbiamo accennato, il sistema di tipi che abbiamo appena presentato è corretto (cioè, accetta solo programmi non-interferenti) ma ammette troppi falsi negativi (cioè, rifiuta troppi programmi che sono non-interferenti). Un semplice esempio è il comando seguente

```
S = P
```

che copia in una variabile segreta il contenuto di una variabile pubblica. Questo chiaramente non costituisce alcun flusso di informazioni rischioso e quindi dovrebbe essere accettato; il sistema di tipi precedentemente presentato, invece, lo rifiuta. Il problema del sistema di tipi appena introdotto è che abbiamo assunto un significato troppo stringente per i giudizi di tipo. Ad esempio, l'intuizione dietro una variabile di tipo τ era di contenero informazioni *esattamente* di livello τ; questa intuizione va indebolita.
Assumiamo quindi che una variabile di livello τ mantenga informazioni di livello τ *o minore*; di conseguenza, una espressione di livello τ sarà formata a partire da informazioni di livello τ *o minore*. Ciò giustifica l'introduzione della regola

$$\frac{\gamma \vdash e : \tau \qquad \tau \leq \tau'}{\gamma \vdash e : \tau'} \tag{3.0.1}$$

Il problema adesso è che il sistema di tipi perde la correttezza; infatti accetta il programma (chiaramente interferente)

```
P = S
```

che copia il valore segreto di S nella variabile pubblica P. Infatti, è possibile inferire il seguente giudizio di tipo:

$$\dfrac{\dfrac{\dfrac{\gamma(P) = L}{\gamma \vdash P : L} \quad L \leq H}{\gamma \vdash P : H} \qquad \dfrac{\gamma(S) = H}{\gamma \vdash S : H}}{\gamma \vdash P = S : Hcmd}$$

Infatti, mentre è giusto poter promuovere una variabile bassa a livello alto in un'espressione tipo $P + S$, è sbagliato fare ciò in un assegnamento del tipo $P = S$. Bisogna quindi distinguere le occorrenze delle variabili a sinistra di un assegnamento da quelle in una espressione: le prime non possono essere promosse, le seconde sì. Modifichiamo pertanto l'insieme dei tipi a disposizione:

Definizione 3.5.1.

Tipi: *Un tipo è definito dalla seguente grammatica:*

$$\rho \ ::= \ \tau \mid \tau var \mid \tau cmd$$

dove $\tau \in \{L, H\}$.

Notiamo che la regola che promuove un'espressione di livello τ a una di livello $\tau' \geq \tau$ non è applicabile al tipo τvar e questo impedisce di tipare l'esempio precedente: il sistema di tipi torna così corretto.
Questo però non basta per accettare il programma (innocuo)

```
S = 0;
P = 0
```

Infatti, a ogni comando va assegnato *un preciso* tipo; quindi, nella composizione sequenziale di due comandi o in un if-then-else dobbiamo avere che i due sottocomandi abbiano lo stesso tipo. Possiamo far ciò in due modi:

$$\frac{\gamma \vdash c : \tau cmd \qquad \tau \leq \tau'}{\gamma \vdash c : \tau' cmd} \tag{3.0.2}$$

$$\frac{\gamma \vdash c : \tau cmd \qquad \tau' \leq \tau}{\gamma \vdash c : \tau' cmd} \tag{3.0.3}$$

Usando la regola (3.0.2) riusciremmo a tipare il programma

```
if (S) { P = 1 } else { P = 0 }
```

che è chiaramente interferente. Pertanto, la strada giusta è assumere (3.0.3).
In realtà, l'uso di (3.0.1) e (3.0.3) complicherebbe leggermente le prove che andremo a fare.[2] Pertanto, assumiamo la seguente formulazione alternativa del sistema di tipi raffinato:

Regola di inferenza 3.5.1.

[2] Il problema sta nel fatto che sia nella premessa che nella conseguenza della regola dobbiamo tipare la stessa espressione/comando. Questo crea ovvi problemi quando procediamo per induzione sulla sintassi dell'espressione/comando, ma è anche problematico quando la prova è per induzione sull'inferenza (in questo caso, non essendo più le regole di tipaggio syntax-driven, non si può dedurre in maniera certa la regola di tipaggio usata guardando solo la sintassi dell'espressione/comando).

$$\frac{}{\gamma \vdash n : \tau} \qquad \frac{\gamma(x) = \tau}{\gamma \vdash x : \tau var}$$

$$\frac{\gamma \vdash x : \tau var \quad \tau \leq \tau'}{\gamma \vdash x : \tau'} \qquad \frac{\gamma \vdash e_1 : \tau \quad \gamma \vdash e_2 : \tau}{\gamma \vdash e_1 \theta e_2 : \tau}$$

$$\frac{\gamma \vdash x : \tau var \quad \gamma \vdash e : \tau \quad \tau' \leq \tau}{\gamma \vdash x = e : \tau' cmd} \qquad \frac{\gamma \vdash c_1 : \tau cmd \quad \gamma \vdash c_2 : \tau cmd}{\gamma \vdash c_1; c_2 : \tau cmd}$$

$$\frac{\gamma \vdash e : \tau \quad \gamma \vdash c_1 : \tau cmd \quad \gamma \vdash c_2 : \tau cmd \quad \tau' \leq \tau}{\gamma \vdash if(e)\ \{c_1\}\ else\ \{c_2\} : \tau' cmd}$$

$$\frac{\gamma \vdash e : \tau \quad \gamma \vdash c : \tau cmd \quad \tau' \leq \tau}{\gamma \vdash while(e)\ \{c\} : \tau' cmd}$$

Si noti ora che ad una variabile x di livello τ si assegna il tipo τvar. Se la variabile è usata a sinistra di un assegnamento, allora questo è proprio il tipo necessario. Se invece la variabile occorre in una espressione, allora il tipo τvar può essere trasformato in un tipo τ ed eventualmente promosso ad un qualsiasi $\tau' \geq \tau$.
Il lettore mostri che il nuovo sistema dei tipi non rigetta l'ultimo programma esposto nella sezione precedente.
Abbiamo anticipato che il sistema di tipi accetta solo programmi non-interferenti, cioè è corretto; purtroppo però non è completo, nel senso che esistono programmi non interferente che però vengono rigettati dal sistema dei tipi. Ad esempio, si consideri

```
if (S) { P = 0 } else { P = 0 }
```

con il tipaggio $\mathcal{L}(P) = L$, $\mathcal{L}(S) = H$. Proviamo a dare tipo $Hcmd$ al comando.

$$\dfrac{\dfrac{\dfrac{\gamma(S) = H}{\gamma \vdash S : Hvar}}{\gamma \vdash S : H} \qquad \dfrac{\dfrac{\gamma(P) \neq H}{\gamma \vdash P : Hvar} \quad \gamma \vdash 0 : H}{\gamma \vdash P = 0 : Hcmd} \qquad \dfrac{\dfrac{\gamma(P) \neq H}{\gamma \vdash P : Hvar} \quad \gamma \vdash 0 : H}{\gamma \vdash P = 0 : Hcmd}}{\gamma \vdash if(S)\ \{P = 0\}\ else\ \{P = 0\} : Hcmd}$$

Come si vede dall'albero delle inferenze, non è possibile dare tipo $Hcmd$ al comando. Se proviamo a dare tipo $Lcmd$ al programma ci blocchiamo immediatamente quando proviamo a dare tipo L alla guardia dell'if. Infatti:

$$\dfrac{\dfrac{\dfrac{\gamma(S) \neq L}{\gamma \vdash S : Lvar}}{\gamma \vdash S : L} \qquad \dfrac{\dfrac{\gamma(P) = L}{\gamma \vdash P : Lvar} \quad \gamma \vdash 0 : L}{\gamma \vdash P = 0 : Lcmd} \qquad \dfrac{\dfrac{\gamma(P) = L}{\gamma \vdash P : Lvar} \quad \gamma \vdash 0 : L}{\gamma \vdash P = 0 : Lcmd}}{\gamma \vdash if(S)\ \{P = 0\}\ else\ \{P = 0\} : Lcmd}$$

Sebbene il programma in questione non abbia alcun flusso di informazione da variabili alte a variabili basse (il valore di P è sempre pari a zero ed è indipendente da quello di S), il programma non è tipabile: il sistema di tipi non è quindi completo. Il problema è che il sistema di tipi dovrebbe decidere se il ramo 'then' ed il ramo 'else' hanno lo stesso effetto sulla memoria, e solo in questo caso accettare il comando. Nel caso in questione il compito sarebbe semplice, ma è facile scrivere comandi in cui tale analisi risulti molto complessa.
Un sistema di tipi completo potrebbe essere definito (basta generare tutte le possibili evoluzioni del programma a partire da una qualunque memoria), ma ciò renderebbe il sistema di tipi non decidibile, e quindi poco utile come tecnica di analisi statica. Tali teniche sono infatti accomunate dal tentativo di conciliare efficienza computazionale e potere di analisi. In generale, si tende ad avere tecniche efficienti ma non complete, e si cerca di minimizzare il numero di falsi negativi prodotti dalla tecnica di analisi.

3.6 Proprietà del sistema dei tipi

In questa sezione, descriviamo alcune proprietà fondamentali del sistema di tipi presentato in Sezione 3.5. Partiamo con il mostrare che la regola di tipaggio intuitiva (3.0.3) è in realtà sussunta dalle regole 3.5.1.

Lemma 3.1. *Se $\gamma \vdash c : \tau cmd$ allora $\gamma \vdash c : \tau' cmd$, per ogni $\tau' \leq \tau$.*

Dimostrazione. Per induzione sulla struttura di c. □

Mostriamo ora formalmente il significato dei tipi assegnabili a espressioni e comandi.

Lemma 3.2 (Simple Security)**.** *Se $\gamma \vdash e : \tau$ allora, per ogni x in e, si ha che $\gamma(x) \leq \tau$.*

Dimostrazione. Per induzione sulla struttura di e. Si hanno due casi base: il caso in cui $e \triangleq n$ è banale; analizziamo il caso in cui $e \triangleq x$. Ricordiamo le regole di inferenza per tale caso:

$$\dfrac{\dfrac{\gamma(x) = \tau'}{\gamma \vdash x : \tau' var} \qquad \tau' \leq \tau}{\gamma \vdash x : \tau}$$

e, banalmente, $\gamma(x) = \tau' \leq \tau$.
Per il caso induttivo, consideriamo solo il caso in cui $e \stackrel{\triangle}{=} e_1 + e_2$. Per ipotesi $\gamma \vdash e : \tau$; allora $\gamma \vdash e_1 : \tau$ e $\gamma \vdash e_2 : \tau$. A questo punto possiamo utilizzare l'ipotesi induttiva su e_1 e e_2; per concludere, basta notare che x appare in e se e soltanto se appare in e_1 oppure in e_2. Gli altri casi sono equivalenti e quindi non verranno esplicitati. □

Il Lemma di *Simple Security* dice che solo le variabili a livello τ o minore saranno lette quando l'espressione e sarà valutata; tale concetto in sicurezza prende il nome di proprietà di *no read up*. Quindi, se $\gamma \vdash e : L$, allora l'espressione e può essere valutata senza leggere alcuna variabile di livello H.

Lemma 3.3 (Confinement)**.** *Se $\gamma \vdash c : \tau cmd$ allora per ogni variabile x assegnata in c si ha che $\gamma(x) \geq \tau$.*

Dimostrazione. Per induzione sulla struttura di c. L'unico caso base da prendere in considerazione è con il comando $x = e$; per definizione del sistema di tipi, deve essere che

$$\frac{\gamma \vdash x : \tau' var \quad \gamma \vdash e : \tau' \quad \tau \leq \tau'}{\gamma \vdash x = e : \tau cmd}$$

dove la prima premessa vale solo se $\gamma(x) = \tau' (\geq \tau)$.
Per il caso induttivo, consideriamo solo il caso in cui $c = while\ (e)\ do\ c'$ (il caso dell'if e della concatenazione sono analoghi). Allora

$$\frac{\gamma \vdash e : \tau' \quad \gamma \vdash c : \tau' cmd \quad \tau \leq \tau'}{\gamma \vdash while(e)\ \{c\} : \tau cmd}$$

Per induzione, $\gamma(x) \geq \tau'$ per ogni x assegnata in c'; poichè le uniche variabili assegnate in c sono le variabili assegnate in c', possiamo concludere. □

Il Lemma di *Confinement* afferma che nessuna variabile al di sotto del livello τ in c viene modificata (*no write down*). In questo modo siamo sicuri che se un comando è di livello H allora nessuna variabile di livello L sarà influenzata durante la computazione di c.

La correttezza di ogni sistema di tipi prevede due passi. Il primo è il risultato di *subject reduction*, in base al quale ogni computazione di programmi ben tipati passa solo per programmi ben tipati. Il secondo è il risultato di *type safety*, secondo il quale un sistema di tipi garantisce sicurezza (nel nostro caso la non-interferenza).

Teorema 3.4 (Subject Reduction)**.** *Se $\gamma \vdash c : \tau cmd$ e $(\mu; c) \rightarrow (\mu'; c')$ allora $\gamma \vdash c' : \tau cmd$.*

Dimostrazione. Dimostriamo la proprietà per induzione sulla struttura del comando. Il caso base è banale; l'unico comando da prendere in considerazione è difatti il comando $c \triangleq x = e$, a seguito del quale il programma termina: dunque, non esiste alcun comando c' tale che $(\mu, c) \rightarrow (\mu', c')$ e la tesi risulta banalmente verificata (una delle ipotesi è falsa e pertanto l'implicazione risulta vera).
Per il caso induttivo è necessario invece analizzare tre casi:

1. $c \triangleq c_1; c_2$. Per ipotesi sappiamo che c è tipabile ($\gamma \vdash c : \tau cmd$) e dunque $\gamma \vdash c_1 : \tau cmd$ e $\gamma \vdash c_2 : \tau cmd$. Analizziamo il caso in cui c' sia uguale a c_2 (il comando c_1 ha richiesto una singola computazione); dall'osservazione precedente sappiamo che c_2 ha tipo τcmd, ma c_2 è uguale a c' che dunque avrà tipo τcmd.

   Supponiamo invece che siano necessari più passi per terminare il comando c_1; avremo dunque: $(\mu; c_1) \rightarrow (\mu'; c_1')$ e $c' = c_1'; c_2$. Per induzione abbiamo che $\gamma \vdash c_1' : \tau cmd$ e quindi $\gamma \vdash c' : \tau cmd$.

2. $c \triangleq if(e)\ \{c_1\}\ else\ \{c_2\}$. Per ipotesi sappiamo che c è tipabile; dunque, $\gamma \vdash e : \tau'$, $\gamma \vdash c_1 : \tau' cmd$, $\gamma \vdash c_2 : \tau' cmd$, per $\tau \leq \tau'$. Per il Lemma 3.1, abbiamo che $\gamma \vdash c_1 : \tau cmd$ e $\gamma \vdash c_2 : \tau cmd$. Poichè c' sarà o c_1 o c_2 (a seconda della valutazione di e), la tesi risulta verificata.

3. $c \triangleq while(e)\ \{c_1\}$. Simile al caso precedente. □

Dimostriamo ora che il sistema di tipi è una tecnica di prova corretta rispetto alla definizione di non-interferenza per flussi impliciti (vedi Definizione 2.2.2).

Teorema 3.5 (Type Safety)**.** *Siano* $\mu =_L \nu$, $\gamma \vdash c : \tau cmd$, $(\mu, c) \rightarrow^* \mu'$ *e* $(\nu, c) \rightarrow^* \nu'$*; allora* $\mu =_L \nu'$.

Dimostrazione. Assumiamo anzitutto che sia possibile inferire $\gamma \vdash c : Hcmd$. Per il Confinement Lemma, $\gamma(x) = H$ per ogni variabile x assegnata in c; quindi, qualunque siano i valori assegnati a qualunque variabile, $\mu' =_L \nu'$.
Assumiamo quindi che non sia possibile inferire $\gamma \vdash c : Hcmd$. Dimostriamo l'asserto per induzione sulla somma delle lunghezze di $(\mu, c) \rightarrow^* \mu'$ e $(\nu, c) \rightarrow^* \nu'$. Ci sono due casi base:

- $c \triangleq x = e$: per definizione della semantica operazionale, deve essere che $(\mu, c) \rightarrow \mu[x \leftarrow m]$ e $(\nu, c) \rightarrow \nu[x \leftarrow n]$, per $(\mu, e) \rightarrow m$ e $(\nu, e) \rightarrow n$. Visto che non è possibile inferire $\gamma \vdash c : Hcmd$, deve essere che $\gamma \vdash c : Lcmd$, da cui $\gamma \vdash x : Lvar$ e $\gamma \vdash e : L$. Per il Simple Security Lemma, e contiene solo variabili basse e su di esse μ e ν coincidono. Pertanto, deve essere $m = n$ e quindi $\mu[x \leftarrow m] =_L \nu[x \leftarrow n]$.

- $c \triangleq while(e)\ \{c'\}$, con $(\mu, c) \rightarrow \mu$ e quindi $\mu' = \mu$; questo può essere solo se $(\mu, e) \rightarrow 0$. Visto che non è possibile inferire $\gamma \vdash c : Hcmd$, deve essere che $\gamma \vdash c : Lcmd$, da cui $\gamma \vdash c' : Lcmd$ e $\gamma \vdash e : L$. Per il Simple Security Lemma, e contiene solo variabili basse e su di esse μ e

ν coincidono. Pertanto, $(\nu, e) \to 0$ e quindi $\nu' = \nu$, da cui la tesi segue banalmente.

Per il passo induttivo, distinguiamo tre casi:

- $c \stackrel{\triangle}{=} \mathit{if}(e)\ \{c_1\}\ \mathit{else}\ \{c_2\}$. Per definizione della semantica operazionale, deve essere che $(\mu, c) \to (\mu, c_i) \to^* \mu'$, per $i = 1$ se $(\mu, e) \to 1$ e $i = 2$ altrimenti. Per ragionamenti simili a quelli nel caso base, la valutazione di e in ν è la stessa che in μ; quindi, $(\nu, c) \to (\nu, c_i) \to^* \nu'$ e possiamo concludere per induzione.

- $c \stackrel{\triangle}{=} c_1; c_2$. Visto che $(\mu, c) \to^* \mu'$, deve essere che $(\mu, c) \to^* (\mu'', c_2) \to^* \mu'$, poichè $(\mu, c_1) \to^* \mu''$. Similmente, visto che $(\nu, c_1) \to^* \nu'$, deve essere che $(\nu, c_1) \to^* \nu''$ e $(\nu'', c_2) \to^* \nu'$. Per induzione abbiamo che $\mu'' =_L \nu''$ e, di nuovo per induzione, $\mu' =_L \nu'$.

- $c \stackrel{\triangle}{=} \mathit{while}(e)\ \{c'\}$. Sia $(\mu, c) \to (\mu, c'; c) \to^* \mu'$; questo può essere solo se $(\mu, e) \to 1$. Per ragionamenti simili a quelli nel caso base, la valutazione di e in ν è la stessa che in μ; quindi, $(\nu, c) \to (\nu, c'; c) \to^* \nu'$. Per induzione si conclude. $\square$

Capitolo 4

Linguaggio Multi-threaded

Nel capitolo precedente è stato sviluppato un sistema di tipi che permette un controllo sul flusso di informazione in un linguaggio imperativo sequenziale. In questo capitolo tratteremo l'estensione del sistema dei tipi a linguaggi di programmazione *multi-threaded*. In seguito, considereremo anche il caso in cui i threads possono avere informazioni sul reciproco tempo di avanzamento, in cui lo scheduling sia regolato (e non puramente non-deterministico) e in cui un'attaccante possa fare osservazioni statistiche sugli output del programma osservato. In tutti questi casi vedremo i tipi di covert channels che si possono originare e opportuni sistemi di tipi per identificarli staticamente.

4.1 Scenario Multi-threaded

Diamo un esempio che mostra che il sistema dei tipi 3.5.1 non rifiuta il seguente programma multi-threaded interferente. Supponiamo che la memoria utilizzata dai diversi thread sia condivisa e sia partizionata in valori alti e bassi. Analizziamo il seguente programma.

Codice 4.1: Programma Multithreaded

```
Thread A:                    Thread B:
  while t0 == 0 do {};         while t1 == 0 do {};
  P = 0;                       P = 1;
  maintrigger++                maintrigger++

        Thread C:
          if S then t0 = 1 else t1 = 1;
          while maintrigger == 0 do {};
          if t0 then t1 = 1 else t0 = 1
```

Supponiamo che tutte le variabili siano state inizializzate a 0 e che la variabile segreta sia **S**. Analizziamo il caso in cui **S** sia pari a 0 o ad 1.

$S \triangleq 1$

1. I due thread **A** e **B** aspettano che il controllo passi al thread **C**; resteranno difatti bloccati sul loro primo comando (**while**);
2. Il thread **C** setta la variabile **t0** ad 1 e si blocca sul ciclo **while**, l'unico thread che può proseguire con la computazione è il thread **A**;
3. Il thread **A** setta la variabile pubblica **P** a 0 e incrementa la variabile maintrigger (ad 1);
4. Il thread **C** riprende il controllo e fa partire il thread **B** settando **t1** ad 1;
5. Il thread **B** setta la variabile pubblica **P** a 1 e incrementa la variabile maintrigger (a 2).

$S \triangleq 0$

1. I due thread **A** e **B** aspettano che il controllo passi al thread **C**; resteranno difatti bloccati sul loro primo comando (**while**);
2. Il thread **C** setta la variabile **t1** ad 1 e si blocca sul ciclo **while**, l'unico thread che può proseguire con la computazione è il thread **B**;
3. Il thread **B** setta la variabile pubblica **P** ad 1 e incrementa la variabile maintrigger (ad 1);
4. Il thread **C** riprende il controllo e fa partire il thread **A** settando **t0** ad 1;
5. Il thread **A** setta la variabile pubblica **P** a 0 e incrementa la variabile maintrigger (a 2).

In entrambi casi è stato copiato il valore della variabile privata **S** nella variabile pubblica **P**. Verifichiamo che, sebbene sia presente un flusso di informazione, il sistema dei tipi 3.5.1 non respinge il programma 4.1. Associamo i seguenti tipi alle variabili:

- **S** = H
- **P** = L
- **t0** = H
- **t1** = H
- **maintrigger** = L

E' possibile dare tipo $Lcmd$ ai thread **A** e **B** mentre tipo $Hcmd$ al thread **C**. Verifichiamo che il programma eseguito dal thread **C** è un $Hcmd$. Analizziamo singolarmente le tre istruzioni; affinchè l'intero programma abbia tipo $Hcmd$ è necessario che tutte le istruzioni siano di tipo $Hcmd$. Iniziamo analizzando il primo if:

$$\dfrac{\dfrac{\dfrac{\gamma(S) = H}{\gamma \vdash S : Hvar}}{\gamma \vdash S : H} \quad \dfrac{\dfrac{\dfrac{\gamma(t0) = H}{\gamma \vdash t0 : Hvar} \quad \gamma \vdash 1 : H}{\gamma \vdash t0 = 1 : Hcmd}}{\gamma \vdash if(S)\ \{t0 = 1\}\ else\ \{t1 = 1\} : Hcmd} \quad \dfrac{\dfrac{\gamma(t1) = H}{\gamma \vdash t1 : Hvar} \quad \gamma \vdash 1 : H}{\gamma \vdash t1 = 1 : Hcmd}}{\gamma \vdash if(S)\ \{t0 = 1\}\ else\ \{t1 = 1\} : Hcmd}$$

Procediamo studiando il caso del while.

$$\dfrac{\dfrac{\gamma(maintrigger) = L}{\gamma \vdash maintrigger : Lvar \qquad L \leq H}}{\dfrac{\gamma \vdash maintrigger : H}{\gamma \vdash while\ (maintrigger)\ do\ \{\} : Hcmd}}$$

Concludiamo con l'ultimo comando. Analizziamo separatamente l'analisi della guardia e le due associazioni.

$$\dfrac{\dfrac{\gamma(t0) = H}{\gamma \vdash t0 : Hvar} \quad \gamma \vdash 1 : H}{\gamma \vdash t0 = 1 : Hcmd} \qquad \dfrac{\dfrac{\gamma(t1\ o\ t0) = H}{\gamma \vdash t1\ o\ t0 : Hvar} \quad \gamma \vdash 1 : H}{\gamma \vdash t1\ o\ t0 = 1 : Hcmd}$$

E dunque

$$\dfrac{\gamma \vdash t0 = 1 : Hcmd \quad \gamma \vdash t1 = 1 : Hcmd \quad \gamma \vdash t0 = 1 : Hcmd}{\gamma \vdash if(t0)\ \{t1 = 1\}\ else\ \{t0 = 1\} : Hcmd}$$

Si lascia al lettore la verifica (banale) per gli altri due thread.

4.2 Sintassi e Semantica Operazionale

Modifichiamo dapprima la sintassi e la semantica operazionale presentate nel capitolo precedente per modellare un linguaggio imperativo con più flussi di esecuzione. I singoli threads sono scritti usando la sintassi di Definizioni 3.1.1 e 3.1.2, e condividono tutti la stessa memoria nel corso della loro esecuzione; per semplicità, assumiamo che l'unica interazione possibile tra thread consista nello scambio di messaggi tramite la memoria condivisa.
Per introdurre i threads nel sistema, assumiamo che ogni thread sia identificato univocamente da un identificatore α e utilizziamo una funzione $\mathcal{O}$ (chiamata *object map*) che associa ad ogni identificatore i comandi del thread corrispondente:

$$\mathcal{O} :\ ThreadId \rightharpoonup Command$$

Per la valutazione delle espressioni e dei comandi utilizziamo le regole della semantica operazionale esposta nella sezione 3.1 aggiungendo le seguenti regole che specificano le transizioni che possono essere effettuate dal sistema di thread.

Regola di inferenza 4.2.1.

$$\dfrac{\mathcal{O}(\alpha) = c \quad (\mu, c) \to \mu'}{(\mu, \mathcal{O}) \to (\mu', \mathcal{O} - \alpha)} \qquad \dfrac{\mathcal{O}(\alpha) = c \quad (\mu, c) \to (\mu', c')}{(\mu, \mathcal{O}) \to (\mu', \mathcal{O}[\alpha \leftarrow c'])}$$

Analizziamo le due regole; in entrambi i casi è in esecuzione il thread con identificatore α che esegue il comando c ($= \mathcal{O}(\alpha)$). Nella prima regola è analizzato il caso in cui il thread α termina la sua esecuzione dopo il comando c mentre nel secondo caso il thread può proseguire con l'esecuzione del comando rimanente (c'). Nel primo caso difatti si elimina il thread con identificatore α dall'insieme di thread da prendere in considerazione e si prosegue dalla memoria ottenuta dalla computazione di c a partire da μ; nel secondo caso si aggiorna il comando associato ad α.
Notiamo che non facciamo nessuna assunzione sulla politica di scheduling per selezionare un thread: il meccanismo modellato dalle regole appena presentate modella uno scheduler non-deterministico, in cui ad ogni istante un qualunque thread non terminato può essere mandato in esecuzione. Studieremo in seguito l'impatto che diverse politiche di scheduling (ad esempio, la politica Round Robin) hanno nell'ambito della non-interferenza.

4.3 Type System

Il sistema dei tipi permette di provare i giudizi di tipo della forma $\gamma \vdash c : \tau cmd$. Anche in questo caso γ denota un *tipaggio*, cioè una funzione parziale da variabili a livelli di sicurezza (per semplicità abbiamo solo H e L). Se $\gamma \vdash c : \tau cmd$ per qualche τ allora diremo che c è *ben tipato in* γ. Diremo che $\mathcal{O}$ è *ben tipato* se esiste un γ tale che $\mathcal{O}(\alpha)$ è ben tipato in γ, per ogni $\alpha \in dom(\mathcal{O})$.
Il sistema di tipi per controllare i flussi di informazione in un ambiente multithreaded differisce dal sistema dei tipi presentanto in 3.5.1 poichè in un ambiente multithreaded si possono costruire covert channels tramite la cooperazione di diversi threads, come mostrato nell'esempio in sezione 4.1. Poichè l'unico modo che diversi threads hanno per sincronizzarsi e cooperare è di sospendersi (tramite un while di corpo vuoto e guardia vera), la modifica principale che dobbiamo apportare al sistema di tipi studiato nel capitolo precedente è di imporre che tutte le guardie dei cicli while siano espressioni di livello basso. Pertanto, l'ultima regola di inferenza di 3.3.2 deve essere modificata come segue:

$$\frac{\gamma \vdash e : L \quad \gamma \vdash c : Lcmd}{\gamma \vdash while(e)\ \{c\} : Lcmd}$$

Chiaramente, questa regola è piuttosto forte, nel senso che molti programmi non-interferenti verranno rigettati per via di essa. D'altro canto, questo sembra essere il modo più semplice per garantire la sicurezza di un programma senza compromettere l'efficienza dell'analisi.
Prima di addentrarci nei dettagli della dimostrazione della correttezza è utile formalizzare la proprietà di non interferenza nei programmi multithreaded. Anche in questo caso si vuole respingere quell'insieme di programmi multithreaded che modificano la memoria bassa in funzione della memoria alta: l'obiettivo è dunque fare in modo che le variabili alte non influenzino il valore delle variabili basse. Definiamo formalmente tale concetto.

Definizione 4.3.1.

Multithreaded Non-Interference: *Un insieme di thread $\mathcal{O}$ è non interferente sse $\forall\mu\forall\nu : \mu =_L \nu$ e $(\mu, \mathcal{O}) \rightarrow^* (\mu', \emptyset)$ si ha che $\mu' =_L \nu'$, per $(\nu, \mathcal{O}) \rightarrow^* (\nu', \emptyset)$.*

Per provare che tutti i programmi ben tipati godono della proprietà di non interferenza mostriamo alcune proprietà chiave del sistema dei tipi.

Lemma 4.1 (Simple Security Multithreaded)**.** *Se $\gamma \vdash e : L$ allora ogni identificatore $x \in e$ ha classe L.*

Dimostrazione. Per induzione sulla struttura di e. □

Lemma 4.2 (Confinement Multithreaded)**.** *Se $\gamma \vdash c : H cmd$ allora ogni identificatore x assegnato in c ha classe H ed è garantito che il comando c termini a partire da qualsiasi memoria μ.*

Dimostrazione. Per induzione sulla struttura di c ed utilizzando il fatto che c non può contenere cicli **while** (i comandi **while** hanno tipo $Lcmd$). □

Teorema 4.3 (Subject Reduction Multithreaded)**.** *Se $\gamma \vdash c : \tau cmd$ e $(\mu, c) \rightarrow (\mu', c')$ allora $\gamma \vdash c' : \tau cmd$.*

Dimostrazione. Per induzione sulla struttura di c. □

Il problema principale nella dimostrazione di non-interferenza che andremo a fare tra breve è che uno stesso insieme di threads valutato in memorie differenti può dar vita ad esecuzioni molto diverse tra loro. Pertanto, risulterà utile definire un'equivalenza tra comandi per poter gestire questo tipo di situazioni.

Definizione 4.3.2.

Due comandi c e d si dicono low-equivalenti (o semplicemente equivalenti), scritto $c =_L d$, se godono di almeno una delle seguenti proprietà:

1. *$c = d$*
2. *c e d hanno tipo $Hcmd$*
3. *$c \stackrel{\triangle}{=} c_1; c_2$, $d \stackrel{\triangle}{=} d_1; d_2$ e $c_1 =_L d_1$ e $c_2 =_L d_2$*

L'intuizione dietro la definizione di comandi low-equivalenti è quella di raggruppare comandi che hanno lo stesso effetto sulla porzione bassa di memorie low-equivalenti; sono cioè comandi che, eseguiti a partire da memorie low-equivalenti, generano ancora memorie low-equivalenti. Chiaramente, comandi uguali sono equivalenti. Comandi alti sono anch'essi low-equivalenti, poichè non accedono a variabili basse e quindi non modificano la parte bassa delle memorie di partenza. Infine, la terza clausola serve per definire equivalenti comandi del tipo $c; d_1$ e $c; d_2$, dove c è un $Lcmd$ e d_1/d_2 sono $Hcmd$.
Abbiamo bisogno di un lemma sull'esecuzione di una composizione sequenziale.

Lemma 4.4. *Se* $(\mu, c_1) \rightarrow^k (\mu', c_1')$*, allora* $(\mu, c_1; c_2) \rightarrow^k (\mu', c_1'; c_2)$.
Se $(\mu, c_1) \rightarrow^k \mu'$*, allora* $(\mu, c_1; c_2) \rightarrow^k (\mu', c_2)$.

Dimostrazione. Entrambi gli enunciati sono dimostrati per induzione su k. Il passo induttivo nella prova del secondo enunciato usa il primo enunciato. □

Teorema 4.5 (Sequential NonInterference)**.** *Siano c e d due comandi ben tipati, μ e ν due memorie, con $c =_L d$ e $\mu =_L \nu$. Se $(\mu, c) \rightarrow (\mu', c')$ allora $(\nu, d) \rightarrow^* (\nu', d')$ tali che $c' =_L d'$ e $\mu' =_L \nu'$. Se $(\mu, c) \rightarrow \mu'$ allora $(\nu, d) \rightarrow^* \nu'$ con $\mu' =_L \nu'$.*

Dimostrazione. Iniziamo analizzando il caso in cui c e d sono degli $Hcmd$. Per il Confinement Lemma, c e d non assegnano nessuna variabile di tipo L.

(i) Se $(\mu, c) \rightarrow (\mu', c')$ allora prendiamo come ν' esattamente la memoria ν e come d' il comando d. Per Subject Reduction, c' ha tipo $Hcmd$, dunque banalemente $c' =_L d'$; ma $\mu' =_L \mu$ e dunque $\mu' =_L \nu'$.

(ii) Se $(\mu, c) \rightarrow \mu'$ allora, appellandoci nuovamente al Confinement Lemma, otteniamo che il comando d sulla memoria ν termina dopo un certo numero di passi; pertanto, $(\nu, c) \rightarrow^* \nu'$ e $\mu' =_L \nu'$.

Trattiamo adesso il caso in cui c e d non siano entrambe di tipo $Hcmd$. La prova è per induzione sulla struttura di c.

- Caso $x = e$ (BASE). Dato che c e d non hanno entrambi tipo $Hcmd$, allora necessariamente $c = d$ ($c =_L d$). Dunque $\gamma(x) = L$ e $\gamma \vdash e : L$. Per il lemma di Simple Security, ogni identificatore in e ha tipo L e viene valutato allo stesso valore n in μ e ν. Dunque $\mu' = \mu[x \leftarrow n] =_L \nu[x \leftarrow n] = \nu'$.

- Caso $c_1; c_2$. Il comando d deve avere la forma $d_1; d_2$ con $d_1 =_L c_1$ e $d_2 =_L c_2$. Se $(\mu, c_1) \rightarrow (\mu', c_1')$, allora per induzione deve esistere (ν', d_1') tale che $(\nu, d_1) \rightarrow^* (\nu', d_1')$ con $c_1' =_L d_1'$ e $\mu' =_L \nu'$. Per il lemma 4.4 $(\nu, d) \rightarrow^* (\nu', d_1'; d_2)$ e possiamo concludere visto che $c_1'; c_2 =_L d_1'; d_2$. Se invece $(\mu, c_1) \rightarrow \mu'$, allora per induzione esiste un ν' tale che $(\nu, d_1) \rightarrow^* \nu'$ con $\mu' =_L \nu'$; di nuovo per il lemma 4.4, $(\nu, d) \rightarrow^* (\nu', d_2)$.

- Caso $if(e)\{c_1\}\{c_2\}$. Dato che c e d non sono entrambi $Hcmd$, allora $c = d$. Inoltre, $\gamma \vdash e : L$ e $\gamma \vdash c_i : Lcmd$; dato che $\mu =_L \nu$, per il simple security lemma, e viene valutata allo stesso valore in μ e ν. Pertanto, $(\mu, c) \rightarrow (\mu, c_i)$ e $(\nu, c) \rightarrow (\nu, c_i)$ e concludiamo banalmente.

- Caso $while(e)\{c'\}$. Simile al precedente. □

Vogliamo adesso estendere il risultato appena ottenuto nel contesto multi threded introdotto. Per farlo dovremo prima provare il seguente lemma:

Lemma 4.6. *Sia $\mathcal{O}(\alpha) = c$; allora*

(a) $(\mu, c) \longrightarrow^k (\mu', c')$ *implica che* $(\mu, \mathcal{O}) \longrightarrow^k (\mu', \mathcal{O}[\alpha \leftarrow c'])$*;*

(b) $(\mu, c) \longrightarrow^k \mu'$ *implica che* $(\mu, \mathcal{O}) \longrightarrow^k (\mu', \mathcal{O} \setminus \{\alpha\})$.

Dimostrazione. La dimostrazione è banale ed avviene per induzione su k. I casi base sono dati dalle regole della semantica operazionale. □

L'ultimo passo richiesto per enunciare il teorema di correttezza del sistema di tipi è quello di formalizzare il concetto di low equivalenza per un dato insieme di threads.

Definizione 4.3.3.

Due insiemi di thrads $\mathcal{O}$ e $\mathcal{O}'$ sono low-equivalenti, scritto $\mathcal{O} =_L \mathcal{O}'$, se e solo se

- $dom(\mathcal{O}) = dom(\mathcal{O}')$, *e*
- $\forall \alpha \in dom(\mathcal{O}).\ \mathcal{O}(\alpha) =_L \mathcal{O}'(\alpha)$.

In base a tale definizione è possibile allora provare la correttezza del sistema di tipi presentato:

Teorema 4.7 (Non-interferenza)**.** *Siano $\mathcal{O}_1 =_L \mathcal{O}_2$ e $\mu =_L \nu$; se $(\mu, \mathcal{O}_1) \rightarrow (\mu', \mathcal{O}_1')$, allora $(\nu, \mathcal{O}_2) \rightarrow^* (\nu', \mathcal{O}_2')$ tale che $\mu' =_L \nu'$ e $\mathcal{O}_1' =_L \mathcal{O}_2'$.*

Dimostrazione. Possiamo distinguere due casi:

$$\begin{array}{ll} \text{(a) } \mathcal{O}_1(\alpha) = c \text{ e } (\mu, c) \rightarrow (\mu', c') & \text{con } \mathcal{O}_1' = \mathcal{O}_1[\alpha \leftarrow c] \\ \text{(b) } \mathcal{O}_1(\alpha) = c \text{ e } (\mu, c) \rightarrow \mu' & \text{con } \mathcal{O}_1' = \mathcal{O}_1 \setminus \{\alpha\} \end{array}$$

Analizzeremo solamente il primo caso, poiché il secondo è del tutto simile. Sia allora $\alpha \in dom(\mathcal{O}_1)$, $\mathcal{O}_1(\alpha) = c$ e $\mathcal{O}_2(\alpha) = d$. Per ipotesi $\mathcal{O}_1 =_L \mathcal{O}_2$, dunque è immediato verificare che $c =_L d$. Inoltre, per il teorema 4.5, si ottiene che $(\nu, d) \rightarrow^* (\nu', d')$ con $\mu' =_L \nu'$ e $c' =_L d'$. Da ciò, per il lemma 4.6, segue che $(\nu, \mathcal{O}_2) \rightarrow^* (\nu', \mathcal{O}_2[\alpha \leftarrow d'])$. Poiché $c' =_L d'$ risulta immediato verificare la relazione $\mathcal{O}_1' =_L \mathcal{O}_2'$.
Il risultato di non-interferenza si ottiene iterando ciò che abbiamo appena dimostrato finchè $O_1' = O_2' = \emptyset$. □

4.4 Il Clock

Molti linguaggi di programmazione includono la possibilità di interagire con il *clock* di sistema; un esempio tipico è il linguaggio JAVA. Ci si aspetta che il clock abbia delle implicazioni sul flusso di informazioni dal momento che informazioni sul tempo sono visibili dai threads e utilizzabili per sincronizzazioni maligne.
Per modellare il *clock* utilizzeremo un identificatore speciale che denoteremo con t; supponiamo per semplicità che il valore di t sia inizialmente pari a 0. Nel nostro modello semplificato il valore di t sarà pari al numero di transizioni che sono state compiute dal sistema: ogni operazione incrementa la variabile t di 1. Mostriamo la regola di inferenza per l'assegnamento; il comportamento delle altre regole è equivalente.

$$\frac{(\mu, e) \rightarrow n}{(\mu, x = e) \rightarrow \mu[x \leftarrow n, t \leftarrow \mu(t) + 1]}$$

Chiaramente, t è accessibile in sola lettura, e quindi non è modificabile dal programma in esecuzione; per far ciò, basta imporre che il tipo di t sia un τ piuttosto che un τvar:

$$\frac{\gamma(x) = \tau}{\gamma \vdash x : \tau var}\, x \neq t \qquad \frac{\gamma(t) = \tau}{\gamma \vdash t : \tau}$$

Assumiamo che la variabile t del clock abbia tipo L e consideriamo il seguente esempio in cui è presente un unico thread. Sia S la variabile alta booleana e P la variabile bassa.

```
if (S) then c

if (t < 10) then { P = 0 } else { P = 1 }
```

Adesso supponiamo che il comando c richieda più di 10 passi; qualsiasi sia il valore di S, esiste uno scheduling che permette di copiare il valore di S in P. Questo è un esempio di *Timing Covered Channe*l in cui lo strumento utilizzato per ottenere informazioni alte è proprio la nuova variabile che identifica il *clock*. E' possibile evitare i Timing Cover Channel assumendo, invece, che $t : H$. Nell'esempio precedente infatti diventa impossibile tipare il programma: il secondo if non può essere un $Hcmd$ poichè i rami sono dei $Lcmd$ nè tanto meno può avere tipo $Lcmd$ poichè la guardia è un H. Un'altra possibilità potrebbe essere quella di forzare i comandi if ad $Hcmd$; ma tale vincolo risulterebbe troppo (e inutilemente) forte.

4.5 Scheduling

La semantica per il linguaggio concorrente (4.2.1) è puramente non deterministica; difatti, ad ogni passo un qualsiasi thread può essere selezionato per l'esecuzione (e questo risulta fondamentale per dimostrare il lemma 4.6, senza

il quale il teorema di non interferenza non si riesce a dimostrare). Imponiamo in questa sezione alcuni vincoli di fairness allo *scheduler* e mostriamo come questo possa rendere falso il lemma 4.6 e, di conseguenza, il teorema di non interferenza.
Studiamo un esempio con due threads:

```
Thread A
if (S) then c;
P = 1

Thread B
P = 0
```

Sia $S : Hvar$, $P : Lvar$ e lo scheduler adotti una politica Round Robin che attiva i threads in ordine alfabetico assegnando un *time slice* maggiore di b. Supponiamo che c sia un $Hcmd$ che impieghi più di b passi per essere eseguito completamente; anche in questo caso, il valore di S è stato copiato in P. Per evitare questo problema, è necessario restringere il tipaggio dei comandi if a $Lcmd$.

$$\frac{\gamma \vdash e : L \quad \gamma \vdash c_1 : Lcmd \quad \gamma \vdash c_2 : Lcmd}{\gamma \vdash if(e)\ \{c_1\}\ else\ \{c_2\} : Lcmd}$$

Si lascia al lettore la verifica che tale restrizione permette di scartare il programma presentato nell'esempio precedente.
Tali restrizioni portano al risultato esposto nel seguente teorema:

Teorema 4.8 (Lockstep Execution)**.** *Se $\mathcal{O}$ è ben tipato, allora $\forall\ \mu, \nu . \mu =_L \nu$ si ha che:*

- *se $(\mu, \mathcal{O}) \rightarrow (\mu', \mathcal{O}')$ allora esiste ν' tale che $(\nu, \mathcal{O}) \rightarrow (\nu', \mathcal{O}')$ e $\mu' =_L \nu'$;*
- *se $(\mu, \mathcal{O}) \rightarrow (\mu', \emptyset)$ allora esiste ν' tale che $(\nu, \mathcal{O}) \rightarrow (\nu', \emptyset)$ e $\mu' =_L \nu'$.*

Il teorema di Lockstep Execution ci permette di garantire la non interferenza anche nel caso in cui si sia specificato un determinato scheduler. Difatti tutte le computazioni a partire dalla memoria μ possono essere effettuate anche utilizzando la memoria low-equivalente ν.

4.6 Non interferenza probabilistica

Uno scenario multithreaded permette anche altri tipi di flussi di informazione, diversi in spirito da quelli presentati finora. Per esempio, assumiamo un scheduling probabilistico, in cui ad ogni passo la scelta del thread da mandare in esecuzione è fatta in base ad una certa distribuzione di probabilità (fissata). Si consideri quindi il seguente programma:

```
Thread A
if (S) { c; c };
P = 1

Thread B
c;
P = 0
```

Assumendo che c sia un $Hcmd$, entrambi i thread di questo programma sono effettivamente tipabili secondo il sistema di tipi introdotto nella sezione 4.3. Vogliamo però studiare la distribuzione di probabilità dei possibili valori che **P** può assumere in base al valore di **S**.
A tale scopo, descriviamo uno scheduling come una stringa di identificatori di threads (genericamente identificati con X), con l'idea che nello scheduling $A\,B\,A\,A\,A\,B\,B$ la prima azione sia stata compiuta da A, la seconda da B, le seguenti tre da A e le ultime due da B. Se $\mathbf{S} = 1$, la probabilità che **P** assuma il valore 1 al termine dell'esecuzione è data dalla probabilità che il thread **B** termini prima del thread **A**; simmetricamente, la probabilità che **P** assuma il valore 0 è la probabilità che **A** termini prima di **B**. Se $\mathbf{S} = 0$, le probabilità sono invertite. Inoltre, se $\mathbf{S} = 1$, il programma multithreaded richiederà $3 * |c| + 3$ operazioni per terminare ($2 * |c| + 2$ per **A** e $|c| + 1$ per **B**), mentre se $\mathbf{S} = 0$ basteranno $|c| + 3$ operazioni (2 per **A** e $|c| + 1$ per **B**). Pertanto, abbiamo che

$$
\begin{aligned}
p\{\mathbf{P} = 1 \mid \mathbf{S} = 1\} &= \frac{|\{X_1 \ldots X_{3|c|+2}\, A\}|}{|\{X_1 \ldots X_{3|c|+3}\}|} \\
p\{\mathbf{P} = 0 \mid \mathbf{S} = 1\} &= \frac{|\{X_1 \ldots X_{3|c|+2}\, B\}|}{|\{X_1 \ldots X_{3|c|+3}\}|} \\
p\{\mathbf{P} = 0 \mid \mathbf{S} = 0\} &= \frac{|\{X_1 \ldots X_{|c|+2}\, B\}|}{|\{X_1 \ldots X_{|c|+3}\}|} \\
p\{\mathbf{P} = 1 \mid \mathbf{S} = 0\} &= \frac{|\{X_1 \ldots X_{|c|+2}\, A\}|}{|\{X_1 \ldots X_{|c|+3}\}|}
\end{aligned}
$$

Se assumiamo che la distribuzione di probabilità associata allo scheduler sia uniforme (ad ogni passo c'è probabilità $\frac{1}{2}$ di eseguire A e, conseguentemente, c'è probabilità $\frac{1}{2}$ di eseguire B), allora possiamo dire che

$$
\begin{aligned}
|\{X_1 \ldots X_{3|c|+2}\, A\}| &= \begin{pmatrix} 3|c|+2 \\ |c|+1 \end{pmatrix} \\
|\{X_1 \ldots X_{3|c|+2}\, B\}| &= \begin{pmatrix} 3|c|+2 \\ |c| \end{pmatrix} \\
|\{X_1 \ldots X_{3|c|+3}\}| &= \begin{pmatrix} 3|c|+3 \\ |c|+1 \end{pmatrix}
\end{aligned}
$$

$$
\begin{aligned}
|\{X_1 \dots X_{|c|+2}\, A\}| &= \begin{pmatrix} |c|+2 \\ |c|+1 \end{pmatrix} \\
|\{X_1 \dots X_{|c|+2}\, B\}| &= \begin{pmatrix} |c|+2 \\ |c| \end{pmatrix} \\
|\{X_1 \dots X_{|c|+3}\}| &= \begin{pmatrix} |c|+3 \\ |c|+1 \end{pmatrix}
\end{aligned}
$$

Pertanto, assumendo che $|c|$ sia grande (già 100 istruzioni sono significative), si ha che

$$
\begin{aligned}
p\{\mathbf{P} = 1 \mid \mathbf{S} = 1\} &= \tfrac{2|c|+2}{3|c|+3} = \tfrac{2}{3} \\
p\{\mathbf{P} = 0 \mid \mathbf{S} = 1\} &= \tfrac{|c|+1}{3|c|+3} = \tfrac{1}{3} \\
p\{\mathbf{P} = 0 \mid \mathbf{S} = 0\} &= \tfrac{|c|+1}{|c|+3} = 1 \\
p\{\mathbf{P} = 1 \mid \mathbf{S} = 0\} &= \tfrac{2}{|c|+3} = 0
\end{aligned}
$$

Ciò che si è mostrato con quest'esempio è che la distribuzione di probabilità del valore assunto dalla variabile **P** è funzione del valore assunto dalla variabile segreta **S**. In particolare, se iterando l'esecuzione si osserva che **P** vale praticamente sempre 0, allora **S** varrà 0; se invece per circa i due terzi delle volte **P** vale 1, allora anche **S** varrà 1. Quindi, un utente che esegue tale programma potrebbe ottenere informazioni sul valore assunto da **S** in base alla distribuzione di probabilità associata al valore che **P** assume al termine dell'esecuzione.
Ciò porta a dover riformulare ancora il concetto di non interferenza. Intuitivamente, un programma è *probabilisticamente non interferente* se e solo se, partendo da memorie low-equivalenti, il programma genera la stessa distribuzione di probabilità sulle memorie low-equivalenti. Ciò implica che la distribuzione di probabilità sui valori in un'arbitraria variabile bassa non dipende dal valore iniziale di una qualsiasi variabile alta. Chiaramente, il sistema di tipi introdotto in sezione 4.5 rigetta il programma appena studiato perchè contiene un if con guardia alta. E' possibile definire un nuovo sistema di tipi meno restrittivo di questo per riconoscere comandi probabilisticamente non interferenti; si rimanda il lettore interessato a [12].

Capitolo 5

Relative Secrecy

5.1 Introduzione

Nei capitoli precedenti abbiamo visto come un programma riesca ad assicurare l'assenza di *information flow* se effettivamente soddisfa la proprietà di *non-interferenza*. Inoltre, dovrebbe risultare chiaro che il desiderio di identificare staticamente programmi non-interferenti porta a forti restrizioni nella scrittura di programmi. Per quanto la segretezza assoluta sia l'obiettivo che si vuole perseguire idealmente, in molte situazioni concrete ci si può accontentare di programmi più deboli. In particolare, nei moderni sistemi di autenticazione (username e password di una casella email, il pin di un bancomat, ecc...) l'aspetto peculiare è il consentire a chiunque di poter eseguire una query e di avere accesso, se il segreto fornito è esatto, oppure di sapere che quel tentativo è fallito. Un'astrazione di tali sistemi è il seguente programma:

```
P = 0

while P != S do P = P + 1
```

E' facile notare che esso rappresenta effettivamente un flusso di informazione: pertanto, nessuno dei sistemi di tipi visti in precedenza potrà accettare un tale programma, che infatti risulta essere interferente.
A questo punto la domanda è: come giustificare l'esistenza di tali sistemi? La risposta si basa sulla complessità del programma: per scoprire un segreto di k bit bisogna far un numero di query esponenziale in k. La *Relative Secrecy*, che andremo ora a studiare, fornisce un sistema di tipi per il quale è possibile dimostrare che nessun programma ben tipato ha un flusso di informazione tale da compromettere i segreti in tempo polinomiale (nella dimensione del segreto). Tuttavia, è indispensabile precisare che ci troviamo in un ambiente in cui il linguaggio è *deterministico* e *single-threaded*.

5.2 Il linguaggio deterministico e il sistema di tipi

Innanzitutto, per semplificare lo scenario che andremo a studiare, modifichiamo leggermente il linguaggio imperativo deterministico presentato nel primo capitolo per modellare in maniera più semplice la possibilità di effettuare query (tramite il costrutto *match*) su una particolare e prefissata variabile alta h.

Definizione 5.2.1.

Espressioni: *Un'espressione è definita dalla seguente grammatica:*

$$e \ ::= \ n \mid x \mid h \mid match(e) \mid e_1 == e_2 \mid e_1 + e_2 \mid e_1 \neq e_2 \mid \cdots$$

dove $x \in \mathcal{V}, n \in Val$. *Sia* $\mathcal{E}$ *l'insieme delle espressioni così definito.*

Definizione 5.2.2.

Comandi: *Un comando è definito dalla seguente grammatica:*

$$c \ ::= \ skip \mid x = e \mid c_1 \ ; \ c_2 \mid if \ (\ e \ ) \ \{ \ c_1 \ \} \ else\{ \ c_2 \ \} \mid while \ (\ e \ )\{ \ c \ \}$$

dove $x \in \mathcal{V}, e \in \mathcal{E}$.

Il linguaggio considerato differisce da quello presentato in precedenza per alcune espressioni e comandi nuovi. Fra le espressioni, h è uno speciale identificatore che rappresenta una variabile alta (il segreto) a cui vogliamo accedere solo tramite matching. Per essere più precisi, però, dobbiamo specificare che questo valore non deve cambiare nel tempo (non è propriamente una variabile) e, pertanto, verrà tipata in modo particolare. L'espressione *match(e)* è, in generale, lo strumento base che ogni attaccante può usare per compromettere il segreto. Nello specifico, è una semplice funzione del tipo *if h = e then ... else ...*, ovvero, confronta il valore dell'espressione col segreto in questione. Questo modella fedelmente ciò che avviene nei reali mezzi di autenticazione. Infine, *skip* è il comando che non effettua alcuna operazione e può essere implementato come $x = x$.

Regola di inferenza 5.2.1.

$$\frac{(\mu, h = e) \to n}{(\mu, match(e)) \to n} \qquad (\mu, skip) \to \mu$$

Il sistema di tipi per le espressioni e i comandi appena visti estende quello di Sezione 3.5 come segue. Per quanto riguarda h, abbiamo già detto che rappresenta una costante, più che una variabile. Perciò, contrariamente al tipaggio di ogni identificatore, verrà tipato con H invece che con $Hvar$. La query abbiamo detto che deve permettere il confronto di una variabile bassa con una alta, inducendo un piccolo flusso di informazione. Ovviamente, se fosse tipata con H non potremmo fare alcun confronto tra h ed espressioni basse; la tipiamo quindi con L. Infine, tipiamo *skip* con τcmd. In conclusione, formalmente avremo:

Regola di inferenza 5.2.2.

$$\frac{\gamma(h) = H}{\gamma \vdash h : H} \qquad \frac{\gamma \vdash e : \tau}{\gamma \vdash match(e) : \tau} \qquad \gamma \vdash skip : \tau cmd$$

5.3 Segretezza Relativa

Teorema 5.1 (Relative secrecy - I)**.** *Non esiste un comando c in grado di copiare il valore di h (a k bits) in una variabile bassa in $O(k^\alpha), \alpha$ costante, se h è acceduto solo tramite match.*

Dimostrazione. Supponiamo, per assurdo, che esista un comando c in grado di copiare il valore di h in una variabile bassa in $p(k)$ passi (dove tale numero di passi è polinomiale in k) accedendo h solo tramite *match*. Scegliamo k sufficientemente grande per cui $2^k > p(k) + 1$. Secondo la nostra ipotesi, il comando può fare al più $p(k)$ tentativi tramite *match*. Segue che $\exists m, n \in \{0, \ldots, 2^k - 1\}$ tali che $m \neq n$ e non sono utilizzati da c in una *match*. Se $h = n$, c deve copiare comunque il valore in una variabile bassa. Se ciò accade, poichè c è deterministico, allora copierà (erroneamente) n nella stessa variabile bassa anche se $h = m$. □

Il lettore dovrebbe notare che, se il comando fosse non-deterministico, allora questo può copiare il valore di h in tempo polinomiale non-deterministico (quindi, il problema è di tipo NP). Ora facciamo la ragionevole assunzione che il segreto sia scelto a caso, cioè sia scelto in base ad una distribuzione di probabilità uniforme.

Teorema 5.2 (Relative secrecy - II)**.** *Se h è distribuita uniformemente su $\{0, \ldots, 2^k - 1\}$, c è un comando che computa in un tempo polinomiale $p(k)$ e che accede h solo tramite match, allora la probabilità di copiare h in una variabile bassa è al più $\frac{p(k)+1}{2^k}$ (k fissato).*

Dimostrazione. Sia $S \subset \{0, \ldots, 2^k - 1\}$ l'insieme dei tentativi che il comando esegue; quindi, la cardinalità di tale insieme è al più $p(k)$. Sia $v \in \{0, \ldots, 2^k - 1\}$ il valore di default che c copia nella variabile bassa se tutti i tentativi hanno avuto esito negativo. Allora, la probabilità di copiare h è:

$$Pr(h \in S \cup \{v\}) = \frac{|S \cup \{v\}|}{2^k} \leq \frac{p(k) + 1}{2^k}$$

□

Cerchiamo ora di studiare la situazione in cui un programma accede al segreto non soltanto tramite *match*. Tale situazione si può ridurre allo scenario considerato finora nel modo seguente: se ad un programma P ben tipato togliamo tutti i comandi tipati con $Hcmd$ che contengono accessi diretti ad h (in generale, a ogni variabile alta) ottenendo il nuovo programma P', che chiameremo simulazione bassa di P), allora possiamo accedere al segreto solo tramite *match* (rivedere le regole di tipaggio); inoltre, se P e P' hanno lo stesso effetto sulla memoria bassa e la stessa complessità temporale, allora P soddisfa la proprietà di relative secrecy formulate nei teoremi 5.1 e 5.2.

Definizione 5.3.1.

Il comando c' è una simulazione bassa (s.b.) del comando c se c' non contiene $Hvar$ e $\forall \mu$ se $(\mu; c) \rightarrow^n \mu'$ allora $(\mu; c') \rightarrow^m \mu''$ dove $\mu' =_L \mu''$ e $m \leq n$.

Lemma 5.3. *Se $(\mu, c_1; c_2) \rightarrow^j \mu'$ allora esiste $0 < k < j$ e μ'' per cui $(\mu, c_1) \rightarrow^k \mu''$ e $(\mu'', c_2) \rightarrow^{j-k} \mu'$.*

Dimostrazione. La dimostrazione è per induzione su j. □

Lemma 5.4. *Se $(\mu, c_1) \rightarrow^j \mu'$ e $(\mu', c_2) \rightarrow^k \mu''$ allora $(\mu, c_1; c_2) \rightarrow^{j+k} \mu''$.*

Dimostrazione. La dimostrazione è per induzione su j. □

Siamo ora pronti per dimostrare la nostra idea iniziale. Per far ciò, utilizzeremo i lemmi appena forniti e alcune nozioni viste in precedenza, in particolare i lemmi di Simple Security e Confinement.

Teorema 5.5. *Se il comando c è ben tipato, allora esiste un comando c' simulazione bassa di c.*

Dimostrazione. 1. caso $\gamma \vdash c : Hcmd$. Poniamo $c' \stackrel{\triangle}{=} skip$. Infatti, per il lemma di Confinement, si ha che $\mu' =_L \mu$ (c assegna solo $Hvar$), dove $(\mu, c) \rightarrow^n \mu'$ e $(\mu, skip) \rightarrow^1 \mu$. Ma $n \geq 1$ e $skip$ lascia la memoria invariata; pertanto c' è simulazione bassa di c.

2. caso $\gamma \vdash c : Lcmd$, c è ben tipato e non gli si può assegnare $Hcmd$. La dimostrazione è per induzione strutturale su c.

   (a) caso $skip$. Abbiamo già detto che $\gamma \vdash skip : Hcmd$, pertanto $skip$ è simulazione bassa di se stesso.

   (b) caso $c \stackrel{\triangle}{=} x = e$. Per quanto detto, c è un $Lcmd$, pertanto deve essere che $\gamma \vdash x : Lvar$ e $\gamma \vdash e : L$. Per Simple Security, ogni variabile in e deve essere una $Lvar$. Perciò, basta prendere $c' \stackrel{\triangle}{=} c$.

   (c) caso $c \stackrel{\triangle}{=} if(e)\{c_1\}else\{c_2\}$. Sempre per quanto detto, c è un $Lcmd$, pertanto $\gamma \vdash e : L$ e $\gamma \vdash c_i : Lcmd$. Per induzione, $\exists c_1', c_2'$ s.b. di c_1, c_2. Affermiamo che la s.b. di c è $c' \stackrel{\triangle}{=} if(e)\{c_1'\}else\{c_2'\}$. Innanzitutto, notiamo che c' non contiene $Hvar$. Se $(\mu, e) \rightarrow 1$ (simile per 0), allora si avrà che $(\mu, c) \rightarrow (\mu, c_1) \rightarrow^{n-1} \mu'$. Poichè c_1' è una s.b. di c_1, allora $\exists \mu''$ t.c. $(\mu, c') \rightarrow (\mu, c_1') \rightarrow^m \mu''$, $m + 1 \leq n$ e le memorie saranno $=_L$.

   (d) caso $c \stackrel{\triangle}{=} c_1; c_2$. Di nuovo, c è un $Lcmd$. Per induzione, $\exists c_1', c_2'$ s.b. di c_1, c_2. Mostriamo che $c' \stackrel{\triangle}{=} c_1'; c_2'$ è una s.b. di c. Supponiamo che $(\mu, c_1; c_2) \rightarrow^n \mu'$. Per il Lemma 5.3, esiste k e μ'' per cui $0 < k < n, (\mu, c_1) \rightarrow^k \mu''$ e $(\mu'', c_2) \rightarrow^{n-k} \mu'$. Poichè c_1' è s.b. di c_1, per induzione abbiamo che $(\mu, c_1') \rightarrow^{t(\leq k)} \nu'' =_L \mu''$ e $(\mu'', c_2') \rightarrow^{m(\leq n-k)} \nu''' =_L \mu'$. Ma c_1 è ben tipato e c_1' ne è una s.b.; quindi, non assegnano h (ricordare che h non è una variabile: è tipata con H e non con $Hvar$). Da ciò possiamo dedurre $\mu''(h) = \mu(h) = \nu''(h)$. Invece, su c_2 e c_2' non possiamo applicare l'induzione poiché partono da memorie diverse. Però, dal fatto che $\nu'' =_L \mu''$ e dal fatto che c_2' non contiene variabili alte, abbiamo che $(\nu'', c_2') \rightarrow^m \nu' =_L \nu''' =_L \mu'$. Per il Lemma 5.4 $(\mu, c_1'; c_2') \rightarrow^{h+m} \nu' =_L \mu'$ e $t + m \leq k + n - k = n$.

   (e) caso $c \stackrel{\triangle}{=} while(e)\{c_1\}$, con c e c_1 che sono $Lcmd$. Per induzione, $\exists c_1'$ s.b. di c_1, quindi affermiamo che $c' \stackrel{\triangle}{=} while(e)\{c_1'\}$ è una s.b. di c. Innanzitutto, in e non vi possono essere $Hvar$, per Simple Security, e c_1' non contiene variabili alte, per definizione di s.b.; pertanto, c' non contiene variabili alte. Supponiamo poi che $(\mu, c) \rightarrow^n \mu'$; per induzione su n, dimostriamo che esistono m e ν' t.c. $(\mu, c') \rightarrow^m \nu'$, con $\nu' =_L \mu'$ e $m \leq n$.

      - Passo base ($n = 1$): in tal caso, per la regola in cui il while termina in un passo, abbiamo $\mu(e) = 0$ e $\mu' = \mu$; basta quindi prendere $m = 1$ e $\nu' = \mu$.

- Passo induttivo ($n > 1$): possiamo riscrivere la regola per il while come $(\mu, c) \to (\mu, c_1; c) \to^{n-1} \mu'$. Per il lemma 5.3 applicato a $(\mu, c_1; c) \to^{n-1} \mu'$, abbiamo che $\exists \mu'', \ k$ t.c. $(\mu, c_1) \to^k \mu''$ e $(\mu'', c) \to^{n-1-k} \mu'$, con $0 < k < n-1$. Per ipotesi induttiva su $(\mu, c_1) \to^k \mu''$, si ha che $\exists \nu'', i$ t.c. $(\mu, c_1') \to^i \nu''$, con $\nu'' =_L \mu''$ e $i \leq k$. Per ipotesi induttiva su $(\mu'', c) \to^{n-1-k} \mu'$, si ha che $\exists \nu''', j$ t.c. $(\mu'', c') \to^j \nu'''$, con $\nu''' =_L \mu'$ e $j \leq n-1-k$. Ora, sappiamo che c_1 è ben tipato e quindi non modifica h; pertanto, $\mu''(h) = \mu(h)$ e $\nu''(h) = \mu(h)$, da cui $\mu''(h) = \nu''(h)$. Poichè c' non accede variabili alte (se non h tramite *match*) e poichè μ'' e ν'' hanno stessi valori nelle variabili basse ed in h, abbiamo che $(\nu'', c') \to^j \nu' =_L \nu''' =_L \mu'$. Per il lemma 5.4, $(\mu, c_1'; c') \to^{i+j} \nu'$. Pertanto, $(\mu, c') \to^{i+j+1} \nu' =_L \mu'$, con $i + j + 1 \leq k + (n - 1 - k) + 1 = n$. □

Dal teorema, poichè un comando basso può accedere il segreto solo tramite *match*, segue:

Corollario 5.6. *Se c è ben tipato allora soddisfa le proprietà di Relative Secrecy (I e II).*

Dimostrazione. Dimostriamo solo il caso I (il caso II è simile e lasciato come esercizio al lettore). Essendo c tipabile, esso ammette una simulazione bassa c' (vedi Teorema 5.5) che, per il Teorema 5.1, non può copiare h in una variabile bassa in $O(k^\alpha)$. Se per assurdo c potesse copiare h in una variabile bassa in $O(k^\alpha)$, per definizione di s.b. avremmo che anche c' potrebbe farlo; contraddizione! □

Per concludere, vogliamo notare che il generico confronto tra un'espressione alta e una bassa va sempre tipato come H, pena la violazione delle proprietà di relative secrecy. Infatti, poichè una $match(e)$ altro non è che una macro per $h == e$, si potrebbe pensare di generalizzare la regola di tipaggio del *match* con la seguente:

$$\frac{\gamma \vdash e_1 : H \qquad \gamma \vdash e_2 : L}{\gamma \vdash e_1 == e_2 : L}$$

Questa intuizione, però, è errata. Per convincersene, si consideri il seguente programma:

```
l = 0;
mask = 2^{k-1};
while mask > 0 do
        b = h & mask;
        if b == mask then
                l = l | mask;
        mask = mask >> 1
```

Tale programma copia in tempo lineare il valore di h nella variabile bassa l. Infatti, si esaminano in successione tutti i bit di h, tramite un *bitwise and* tra h e la maschera $mask$, copiandoli a turno in b. Questo è un flusso di informazione, poichè scopriamo ad ogni iterazione un bit del segreto: basta settare il bit corrispondente di l a b, nel caso in cui b sia 1; per far ciò, basta controllare il valore di b. In particolare, se b è uguale al bit corrispondente della maschera, allora il bit corrispondente di h è pari a 1 ed un semplice *bitwise or* ci permette di settare in l tale bit. Nel caso in cui b non fosse uguale al corrispondente bit della maschera (il bit del segreto è 0), poichè tutti i bit di l sono inizializzati a 0, contiene già il valore corretto in quella posizione. Il programma termina dopo k iterazioni del while, il cui corpo ha complessità costante. Inoltre, con la regola di tipaggio proposta, il programma risulterebbe tipabile, assumendo che $\gamma(l) = \gamma(mask) = \gamma(k) = L$ e $\gamma(h) = \gamma(b) = H$.

Infine, notiamo che nei moderni sistemi vi è un file che non contiene le password vere e proprie, ma l'hash (one-way) di queste. Anche in questo caso è possibile, attraverso opportuni accorgimenti, rientrare nell'ambito della relative secrecy. In particolare, aggiungiamo la funzione $hash$ e una variabile legata all'immagine del segreto h, chiamiamola fh. Ovviamente dobbiamo avere che $\mu(fh) = \mu(hash(h))$. Inoltre, poniamo

$$\gamma \vdash fh : L \qquad \frac{\gamma \vdash e : \tau}{\gamma \vdash hash(e) : \tau}$$

La funzione $hash$ non può essere sempre tipata L, altrimenti si potrebbe copiare h in tempo lineare con un programma simile al precedente:

```
l = 0;
mask = 2^{k-1};
while mask > 0 do
    if hash(mask) == hash(mask & h) then l = l | mask;
    mask = mask >> 1
```

Si noti che, con questo programma, non abbiamo la certezza assoluta che ogni bit del segreto venga correttamente copiato per via di possibili collisioni in $hash$; ma questo però è poco probabile se $hash$ è implementata bene.
Infine, si può notare come la variabile fh rende l'espressione $match$ praticamente inutile, poichè ora è possibile sostituire l'espressione $match(e)$ (che, ricordiamo, stava per $e == h$) con $hash(e) == fh$; di nuovo, ci possono essere collisioni e quindi le cose non sono esattamente equivalenti. Trasformando le cose in questo modo, possiamo asserire che la difficoltà di copiare il valore del segreto, utilizzando soltanto riferimenti alla variabile fh e chiamate alla funzione $hash$, non è inferiore alla difficoltà di invertire la funzione stessa. Infatti, diciamo per assurdo che esista un programma c tipabile in grado di copiare h in una variabile bassa in $p(k)$, dove k è la dimensione del segreto. Per il Teorema 5.5 esiste c' s.b. di c, che quindi è in grado di copiare h in una variabile bassa in al più $p(k)$; inoltre, essendo c' una s.b., esso conterrà solo invocazioni di $hash$ su espressioni

basse e userà al più fh. Quindi, c' sarebbe (l'implementazione di) un algoritmo in grado di invertire la funzione $hash$ (infatti, c è in grado di copiare il valore di h qualunque sia tale valore).

Capitolo 6

Interferenza Quantificata

6.1 Introduzione

L'analisi statica di un programma, come vista in precedenza, ammette dei falsi negativi e pone dei vincoli sulla sintassi; ad esempio il comando

if S=1 then P=0 else P=0

dove S e P sono rispettivamente alta e bassa, sarebbe rigettata da tutti i sistemi di tipi esaminati. Analizzando meglio il codice si vede però che la variabile P assume sempre il valore 0 indipendentemente da S; quindi in questo caso avremmo un'istruzione non permessa dal sistema dei tipi dove non c'è flusso di informazioni.
Ciò che presenteremo in questo capitolo non è più un'analisi assoluta del codice ma un'analisi *quantitativa*, in grado di misurare quanti bit riusciamo a conoscere delle variabili alte. Un modo molto ragionevole di indebolire l'analisi statica consiste nel misurare quanta informazione fluisce dalle variabili alte alle basse e stabilire se tale flusso è quantitativamente accettabile o meno. Per fare questo ci verranno in aiuto alcune nozioni di Teoria dell'Informazione, che è una disciplina finalizzata a fornire varie misure della quantità di informazione.

6.2 Informazione e Informazione condizionata (richiami)

Definizione 6.2.1.

Una variabile aleatoria discreta X è una funzione $X : S \to \Re$ dove S è uno spazio di probabilità campionario finito o numerabile.

Definiamo la probabilità che una variabile aleatoria X possa assumere un valore x come

$$P(X = x) = \sum_{s \in S : X(s) = x} P(s)$$

Siano $X_1, \ldots, X_n$ variabili aleatorie sullo stesso spazio di probabilità S (cioè $X_i : S \to \Re$, per ogni i); definiamo un *vettore di variabili aleatorie* $< X_1, \cdots, X_n >$ che è una nuova variabile aleatoria da $S \times \ldots \times S \to \Re \times \ldots \times \Re$ tale che

$$< X_1, \cdots, X_n > (s) =< X_1(s), \cdots, X_n(s) >$$

Definizione 6.2.2.

Siano X e Y variabili aleatorie discrete, allora X e Y sono indipendenti sse $P(X = x, Y = y) = P(X = x)P(Y = y)$.

Definizione 6.2.3.

L'entropia di una variabile aleatoria X è data da

$$\mathcal{H}(X) = \sum_{x \in \Re} P(X = x) \log_2 \frac{1}{P(X=x)} \ ([a])$$

[a] Definiamo $0 \cdot \log_2 \frac{1}{0} = 0$ in quanto $\lim_{x \to 0} x \cdot \log_2 \frac{1}{x} = 0$

Intuitivamente, l'entropia di una variabile aleatoria X, indicata con $\mathcal{H}(X)$, è la misura dell'incertezza di un osservatore che osserva una variabile X conoscendo la sua distribuzione di probabilità ovvero è il numero medio di bit ignoti del valore di X. Chiariamo questo concetto tramite alcuni esempi.

Esempio

Sia X una variabile aleatoria discreta a valori di k bit. Consideriamo i seguenti casi:

1. Sia X una variabile aleatoria costante, cioè $P(X = n) = 1$ per un certo n e $P(X = m) = 0$ per ogni $m \neq n$; allora, $\mathcal{H}(X)$=0 perchè sappiamo tutto della variabile X: i bit ignoti di X sono 0.
2. Sia X distribuita uniformemente su $\{0..2^k - 1\}$, cioè $P(X = n) = \frac{1}{2^k}$ per ogni n; allora, $\mathcal{H}(X)$=k: i bit ignoti di X sono k.

3. Sia X distribuita uniformemente su $\{0..2^{k-1}-1\}$), cioè

$$P(X=n) = \begin{cases} 0 & \text{se } x > 2^{k-1} \\ \frac{1}{2^k-1} & \text{altrimenti} \end{cases}$$

   Allora $\mathcal{H}(X)=k-1$.

4. Sia X distribuita uniformemente su $\{0..2^{\frac{k}{2}}-1\}$), cioè

$$P(X=n) = \begin{cases} 0 & \text{se } x > 2^{\frac{k}{2}} \\ \frac{1}{2^{\frac{k}{2}}} & \text{altrimenti} \end{cases}$$

   Allora $\mathcal{H}(X)=\frac{k}{2}$.

Per quanto riguarda l'entropia, quello che veramente conta di una variabile aleatoria X è la partizione che essa induce sull'insieme S, definita come segue. Essendo X una funzione, possiamo sempre definire l'equivalenza $\sim_X$ tale che $a \sim_X b$ se e soltanto se $X(a) = X(b)$.

Definizione 6.2.4.

X e Y inducono la stessa partizione sul loro dominio S, scritto $X \simeq Y$, sse $S/_{\sim_X} = S/_{\sim_Y}$.

L'entropia di variabili che inducono la stessa partizione è uguale.

Proprietà 6.1. *Se $X \simeq Y$ allora $\mathcal{H}(X) = \mathcal{H}(Y)$*

Dimostrazione. Sia B un blocco e $s \in B$; sia inoltre $X(s) = x$ e $Y(s) = y$. Allora

$$P(X=x) = \sum_{s \in B} P(s) = P(Y=y)$$

da cui

$$P(X=x) \cdot \log \frac{1}{P(X=x)} = P(Y=y) \cdot \log \frac{1}{P(Y=y)}$$

Il blocco B è identificabile da x, se è inteso come parte di $S/_{\sim_X}$ ed è identificabile da y, se è inteso come parte di $S/_{\sim_Y}$; pertanto

$$\sum_{x \in Im(X)} P(X=x) \log \frac{1}{P(X=x)} = \sum_{y \in Im(Y)} P(Y=y) \log \frac{1}{P(Y=y)}$$

da cui $\mathcal{H}(X) = \mathcal{H}(Y)$. □

Prima di continuare con altre definizioni, presentiamo alcune importanti proprietà dell'entropia.

Proprietà 6.2.

1. $\mathcal{H}(X) \geq 0$*; l'uguaglianza si ha sse la distribuzione di probabilità associata a* X *è banale (degenere).*
2. $\mathcal{H}(X) \leq \lceil \log |Im(X)| \rceil$*; l'uguaglianza si ha sse la distribuzione di probabilità associata a* X *è uniforme.*
3. $\mathcal{H}(X,Y) \leq \mathcal{H}(X)+\mathcal{H}(Y)$*; l'uguaglianza si ha sse* X *e* Y *sono indipendenti.*

Dimostrazione. Il primo fatto discende immediatamente dal fatto che $P(X = x) \in [0,1]$ e quindi $\log \frac{1}{P(X=x)} \geq 0$; l'uguaglianza si ha solo quando tutti i termini nella somma che definisce l'entropia sono nulli, cioè con la distribuzione degenere (è facile dimostrare che, se esistono $x \neq y$ tali che $P(X = x), P(X = y) > 0$ allora $\mathcal{H}(X) > 0$).
Il secondo fatto discende dal fatto che $\mathcal{H}(X)$ misura l'incertezza (espressa in numero di bit) riguardo X; pertanto, la tesi segue dal fatto che i valori che X può assumere sono codificabili con $\lceil \log |Im(X)| \rceil$. Inoltre, nel caso in cui X segua una distribuzione uniforme, $P(X = x) = \frac{1}{|Im(X)|}$, per ogni $x \in Im(X)$.
Il terzo fatto si dimostra a partire dalle seguenti osservazioni: $P(X = x, Y = y) = P(X = x)P(Y = y|X = x)$; $\log \frac{1}{ab} = \log \frac{1}{a} + \log \frac{1}{b}$; $P(X = x) = \sum_{y \in Im(Y)} P(X = x, Y = y)$; $P(Y = y|X = x) \geq P(Y = y)$, con l'uguaglianza se e solo se X e Y sono indipendenti, da cui $\log \frac{1}{P(Y=y|X=x)} \leq \log \frac{1}{P(Y=y)}$. □

Un'altra definizione base è quella dell'entropia condizionata, che intuitivamente misura l'incertezza su X conoscendo Y.

Definizione 6.2.5.

Entropia condizionata: $\mathcal{H}(X\|Y) \stackrel{\triangle}{=} \mathcal{H}(X,Y) - \mathcal{H}(Y)$

Proprietà 6.3.

1. $\mathcal{H}(X|Y) \geq 0$*; l'uguaglianza si ha sse la distribuzione di probabilità associata a* X *è funzione di* Y*.*
2. $\mathcal{H}(X|Y) \leq \mathcal{H}(X)$*; l'uguaglianza si ha sse* X *e* Y *sono indipendenti.*

Dimostrazione. Per dimostrare la prima proprietà, basta dimostrare che $\mathcal{H}(Y|X) = \sum_x \sum_y P(X = x, Y = y) \log \frac{1}{P(Y=y|X=x)}$ e osservare che ogni addendo è non negativo; inoltre, non è difficile vedere che $\mathcal{H}(X,Y) = \mathcal{H}(Y)$ se e soltanto se X è funzione di Y. La seconda proprietà discende direttamente dalla proprietà 6.2(3). □

E' spesso utile avere l'informazione opposta, cioè la quantità di informazione su una variabile aleatoria che può essere ricavata osservandone un'altra. La *mutua informazione* misura proprio questa quantità.

Definizione 6.2.6.

Mutua informazione: $\mathcal{I}(X,Y) \stackrel{\triangle}{=} \mathcal{H}(X) - \mathcal{H}(X|Y)$

Proprietà 6.4.

1. $\mathcal{I}(X,Y) \geq 0$*; l'uguaglianza si ha sse* X *e* Y *sono indipendenti.*
2. $\mathcal{I}(X,Y) \leq \mathcal{H}(X)$*; l'uguaglianza si ha sse* X *è funzione di* Y*.*
3. $\mathcal{I}(X,Y) = \mathcal{I}(Y,X)$

Dimostrazione. Si noti che la prima e la seconda proprietà discendono direttamente dalle proprietà 6.3(2) e (1), rispettivamente. La terza proprietà, invece, discende direttamente dalla definizione di mutua informazione. Infatti, $\mathcal{I}(X,Y) = \mathcal{H}(X) + \mathcal{H}(Y) - \mathcal{H}(X,Y)$. □

Come per l'entropia, esiste anche la mutua informazione condizionata.

Definizione 6.2.7.

Mutua Informazione Condizionata: $\mathcal{I}(X,Y|Z) \stackrel{\triangle}{=} \mathcal{H}(X|Z) + \mathcal{H}(Y|Z) - \mathcal{H}(X,Y|Z)$

Gli strumenti appena definiti saranno utili per quantificare il flusso di informazione da variabili alte a variabili basse.

6.3 Misura del flusso di informazione

Considereremo il flusso di informazione in sistemi finiti; per modellare tali sistemi in maniera astratta, considereremo un programma come una variabile aleatoria che trasforma una memoria di input in una di output. Inoltre, vogliamo anche modellare diverse distribuzioni di probabilità sui possibili input del sistema ed il conseguente comportamento del programma su tali input. In generale, il programma potrà scegliere non-deterministicamente (o probabilisticamente) il suo comportamente; ci restringeremo più avanti a caso deterministico per dimostrare come la misura che introdurremo sia in accordo con la nozione di non-interferenza vista nel primo capitolo.

Definizione 6.3.1.

Un sistema trasformazionale è una tripla $(\mathcal{S}, \mathcal{I}_\mathcal{S}, \mathcal{O}_\mathcal{S})$ in cui il primo elemento è uno spazio di probabilità su un insieme finito; il secondo è una variabile aleatoria $\mathcal{I}_\mathcal{S} : \mathcal{S} \to \mathcal{I}$ in cui l'insieme delle immagini rappresenta i valori di input; infine, il terzo è una variabile aleatoria $\mathcal{O}_\mathcal{S} : \mathcal{S} \to \mathcal{O}$ in cui l'insieme delle immagini rappresenta i valori di output.

Per fare un esempio, consideriamo il lancio di una moneta che con uguale probabilità dia 0 o 1. Sia $\Sigma \stackrel{\triangle}{=} Var \times \mathcal{N}$ l'insieme di tutte le memorie, con $\mu \in \Sigma$. Definiamo:

$$\mathcal{S} \stackrel{\triangle}{=} \Sigma \times \Sigma \qquad \mathcal{I}_\mathcal{S}(\mu, \mu') = \mu \qquad \mathcal{O}_\mathcal{S}(\mu, \mu') = \mu'$$

e sia P una distribuzione di probabilità su Σ. Il programma che consideriamo è

```
y = coin
```

Si ricava che la distribuzione di probabilità su $\Sigma \times \Sigma$ a partire da P è:

$$\begin{array}{c} \text{se } \mu' = \mu[y \leftarrow 0] \text{ allora } Pr(\mu, \mu') = \frac{P(\mu)}{2} \\ \text{se } \mu' = \mu[y \leftarrow 1] \text{ allora } Pr(\mu, \mu') = \frac{P(\mu)}{2} \\ \text{altrimenti } Pr(\mu, \mu') = 0 \end{array}$$

Introduciamo nel sistema alcune funzioni suriettive chiamate *osservazioni*, il cui scopo è modellare il fatto che in generale non tutta la memoria è osservabile (o, nel nostro caso, interessante: nel definire la non-interferenza a noi interessa solo osservare input alti e output bassi):

$X : \mathcal{I} \to \mathcal{X}$ osservazione di input
$Y : \mathcal{O} \to \mathcal{Y}$ osservazione di output

Tali osservazioni inducono due variabili aleatorie:

$$X^{in} : \mathcal{S} \to \mathcal{X}, \text{ con } X^{in} \stackrel{\triangle}{=} X \circ \mathcal{I}_\mathcal{S}$$

$$Y^{out} : \mathcal{S} \to \mathcal{Y}, \text{ con } Y^{out} \stackrel{\triangle}{=} Y \circ \mathcal{O}_\mathcal{S}$$

Un punto di partenza per controllare il flusso di informazioni dagli input agli output nel sistema potrebbe essere quello di considerare la mutua informazione fra le due variabili aleatorie:

$$\mathcal{I}(X^{in}, Y^{out})$$

Tale definizione è però non appropriata: supponiamo che $Y = XxorZ$, dove X e Z sono variabili aleatorie binarie indipendenti e uniformemente distribuite. Poichè Y è funzione delle altre due, ci aspettiamo che vi sia un flusso di informazione, ad esempio da X a Y, quindi dovremmo ottenere che

$\mathcal{I}(X^{in}, Y^{out}) > 0$. Invece, applicando la definizione di mutua informazione, otteniamo (spesso trascureremo gli apici *in* e *out* quando questi sono chiari dal contesto):

$$\mathcal{I}(X^{in}, Y^{out}) = \mathcal{H}(X) + \mathcal{H}(Y) - \mathcal{H}(X, Y) = \log 2 + \log 2 - \log 4 = 1 + 1 - 2 = 0$$

Infatti,

- l'entropia di X è $2 \cdot \frac{1}{2} \cdot \log_2 2 = 1$;
- un calcolo analogo vale per l'entropia di Y, visto che $Pr(Y = 0) = Pr(X = 0, Z = 0) + Pr(X = 1, Z = 1) = 2 \cdot \frac{1}{2} \cdot \frac{1}{2}$ e simile per $Pr(Y = 1)$;
- infine, $Pr(X = 0, Y = 0) = Pr(X = 0) \cdot Pr(Y = 0|X = 0) = Pr(X = 0) \cdot Pr(Z = 0) = \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4}$ e analogo per tutti gli altri casi, da cui $\mathcal{H}(X, Y) = 4 \cdot \frac{1}{4} \cdot \log_2 4 = 2$.

Il problema si spiega facilmente pensando a Z come alla chiave utilizzata per criptare X: da questo punto di vista, appare evidente come l'osservatore non possa conoscere nulla di X da Y senza conoscere Z. Pertanto, nel calcolare quanto di X sappiamo osservando Y non possiamo trascurare il valore di Z. Se il valore iniziale di Z è ignoto, allora potremmo calcolare

$$\begin{aligned} \mathcal{I}(< X^{in}, Z^{in} >, Y^{out}) &= \mathcal{H}(X, Z) + \mathcal{H}(Y) - \mathcal{H}(X, Y, Z) \\ &= \mathcal{H}(Y) - \mathcal{H}(Y|X, Z) = \mathcal{H}(Y) = 1 \end{aligned}$$

dove la penultima uguaglianza si ha grazie alla proprietà 6.3(1), essendo Y funzione di X e Z. Infatti, osservando Y possiamo sapere se X e Z sono uguali o meno: 1 bit di informazione (dei due ignoti) è rivelato. Se invece il valore di Z è noto, allora potremmo calcolare

$$\mathcal{I}(X^{in}, < Y^{out}, Z^{in} >) = \mathcal{H}(X) - \mathcal{H}(X|Y, Z) = \mathcal{H}(X) = 1$$

e quindi, com'è ovvio, conoscendo Y e Z sappiamo tutto di X (essendo X funzione di Y e Z).
Questo esempio ci insegna che *tutta* la memoria (rilevante per il programma) va tenuta in considerazione quando si cerca di misurare il flusso di informazione. Ad esempio, la parte bassa della memoria di input è osservabile e nota all'attaccante. Pertanto, un miglioramento della definizione precedente consiste nel considerare la mutua informazione condizionata (dove, in alcuni casi, il condizionamento potrà risultare vuoto):

$$\begin{aligned} \mathcal{I}(X^{in}, Y^{out}|Z^{in}) &= \mathcal{H}(X|Z) + \mathcal{H}(Y|Z) - \mathcal{H}(X, Y|Z) \\ &= \mathcal{H}(X|Z) + \mathcal{H}(Y, Z) - \mathcal{H}(Z) - \mathcal{H}(X, Y, Z) + \mathcal{H}(Z) \\ &= \mathcal{H}(X) - \mathcal{H}(X|Y, Z) \\ &\quad \text{per proprietà 6.3(1), essendo } X \text{ funzione di } Y \text{ e } Z \\ &= 1 - 0 \\ &= 1 \end{aligned}$$

Per convincersi della bontà di tale definizione come misura del flusso di informazione, consideriamo qualche altro esempio. Supponiamo che x venga scelta a caso nell'insieme $\{-2^{k-1}+1, \ldots, 2^{k-1}-1\}$, ovvero x è un intero a k bit in complemento a 2. Consideriamo il seguente programma:

```
z = abs(x);
if x == 0 then y = 0 else y = 1
```

Il flusso è quantificabile come segue:

$$\begin{aligned}
\mathcal{I}(X^{in}, <Y^{out}, Z^{out}>) &= \mathcal{H}(X) - \mathcal{H}(X|Y,Z) \\
&= \mathcal{H}(X) - \mathcal{H}(X|Z) \\
&\quad \text{se si conosce } Z \text{ allora si conosce } Y \\
&= \mathcal{H}(X) - \mathcal{H}(X,Z) + \mathcal{H}(Z) \\
&= \mathcal{H}(X) - \mathcal{H}(Z,X) + \mathcal{H}(Z) \\
&= \mathcal{H}(Z) - \mathcal{H}(Z|X) \\
&= \mathcal{H}(Z) \sim k-1 \qquad \text{bit di informazione} \\
&\quad \text{per proprietà 6.3(1), essendo } Z \text{ funzione di } X
\end{aligned}$$

Per $x = 0$ sappiamo tutto, mentre per $x \neq 0$ non sappiamo il segno.
Consideriamo ora il seguente programma:

```
z = abs(x);
y = sgn(x)
```

Ci aspettiamo che il flusso di informazione sia completo, e infatti

$$\begin{aligned}
\mathcal{I}(X^{in}, <Y^{out}, Z^{out}>) &= \mathcal{H}(X) - \mathcal{H}(X|Y,Z) \\
&= \mathcal{H}(X) \\
&\sim k
\end{aligned}$$

Consideriamo infine:

```
z = sgn(x);
if x == 0 then y = 0 else y = 1
```

Il flusso di informazione in questo caso è

$$\begin{aligned}
\mathcal{I}(X^{in}, <Y^{out}, Z^{out}>) &= \mathcal{H}(X) - \mathcal{H}(X|Y,Z) \\
&= \mathcal{H}(X) + \mathcal{H}(Y,Z) - \mathcal{H}(X,Y,Z) \\
&= \mathcal{H}(Y,Z) - \mathcal{H}(Y,Z|X) \\
&= \mathcal{H}(Y,Z) \\
&\quad \text{per proprietà 6.3(1), essendo } Y \text{ e } Z \text{ funzioni di } X
\end{aligned}$$

E' ora necessario calcolare esplicitamente i 4 casi possibili:

$$\begin{aligned} P(y=0,z=0) &= P(x=0) \sim \frac{1}{2^k} \\ P(y=0,z=1) &= 0 \\ P(y=1,z=0) &= P(x>0) \sim \frac{1}{2} \\ P(y=1,z=1) &= P(x<0) \sim \frac{1}{2} \end{aligned}$$

da cui

$$\mathcal{H}(Y,Z) \sim \frac{k}{2^k} + \frac{1}{2} + \frac{1}{2} = 1 + \frac{k}{2^k}$$

Ciò vuol dire che, come era da aspettarsi, per k grande, il flusso è praticamente limitato al solo bit di segno.

6.4 Non interferenza

In questa sezione consideriamo il concetto di non interferenza in sistemi deterministici, dove cioè gli output sono funzione degli input; più precisamente, dato un sistema trasformazionale $(\mathcal{S}, \mathcal{I}_\mathcal{S}, \mathcal{O}_\mathcal{S})$, abbiamo che $O_S = f \circ I_S$. Sia gli input che gli output del sistema possono essere visti come vettori di osservazioni ovvero possiamo assumere l'esistenza di un insieme $\{X_1, \cdots, X_n\}$ t.c. $I_S =< X_1, \cdots, X_n >^{in}$ e un insieme $\{Y_1, \cdots, Y_n\}$ t.c. $O_S =< Y_1, \cdots, Y_n >^{out} = f \circ < X_1, \cdots, X_n >^{in}$.
In precedenza abbiamo definito non-interferente un programma in cui input alti non influenzano output bassi; cerchiamo ora una formulazione più astratta di tale concetto.

Definizione 6.4.1.

Date due relazioni R e S, una funzione f si dice che mappa R in S, scritto $f : R \Rightarrow S$, sse $\forall (x, x') \in R.(f(x), f(x')) \in S$.

In seguito, scriveremo che due memorie μ e μ' sono equivalenti rispetto ad un dato insieme di variabili X come $\mu =_X \mu'$ definito da $\forall x \in X.\mu(x) = \mu'(x)$. Restingiamo ora le osservazioni di input e di output solo a variabili alte e a variabili basse. Possiamo così definire il concetto di non interferenza come segue:

Definizione 6.4.2.

Una funzione f è non interferente se $f :=_L \Rightarrow =_{L'}$, dove L e L' sono rispettivamente gli insiemi di variabili basse di input e di output.

Viceversa, definiamo la presenza di un flusso di informazioni da una certa zona di memoria ad un'altra.

Definizione 6.4.3.

Sia A un'osservazione di input e B un'osservazione di output; diremo che A e B interferiscono a seguito di f sse $f :=_{\overline{A}} \not\Rightarrow =_B$, dove con $\overline{A}$ si intende il complemento di A, cioè tutti gli input che non sono in A.

Nel nostro caso siamo interessati a variabili alte, indicate con H, e a variabili basse, indicate con L. Applicando la definizione 6.4.3 otteniamo che $\overline{H} = L$ e H interferisce con L quando $f :=_L \not\Rightarrow =_{L'}$.
Per dimostrare che la misura di flusso di informazione che abbiamo deciso di usare rispetta tale nozione di non-interferenza, abbiamo bisogno di qualche notazione e risultato preliminare. Sia R una qualsiasi relazione di equivalenza su un insieme D; indichiamo con D/R l'insieme delle classi di equivalenza di D rispetto a R. Definiamo inoltre la funzione $[\_]_R : D \to D/R$ come la funzione che dato un elemento di D restituisce la sua classe di equivalenza, cioè $[d]_R = \{d' \in R \,|\, (d, d') \in R\}$.

Definizione 6.4.4.

Data una variabile aleatoria $X : D \to \mathbf{R}$ il kernel di X è definito come la relazione di equivalenza su D, indicata con ker(X), tale che $(d, d') \in ker(X)$ sse $X(d) = X(d')$.

Si noti che, precedentemente in questo capitolo, avevamo denotato il $ker(X)$ con $\sim_X$ e scrivevamo $X \simeq Y$ ogni volta che $ker(X) = ker(Y)$ (si veda, ad esempio, la Proposizione 6.1). Definiamo ora delle funzioni che useremo nel resto del capitolo:

$$var^{in}(R) \triangleq [\_]^{in}_R \triangleq [\_]_R \circ I_S$$
$$var^{out}(R) \triangleq [\_]^{out}_R \triangleq [\_]_R \circ O_S \triangleq [\_]_R \circ f \circ I_S$$

Spesso ometteremo gli apici *in* e *out* quando saranno irrilevanti o chiari dal contesto.

Lemma 6.5. *$ker(var(R)) = R$ e $var(ker(X)) \simeq X$.*

Dimostrazione. Per la prima uguaglianza, notiamo che $(d, d') \in R$ se e soltanto se $[d]_R = [d']_R$, da cui $var(R)(d) = var(R)(d')$ che, per definizione, equivale a $(d, d') \in ker(var(R))$. Per la seconda uguaglianza, notiamo che $var(ker(X))(d) = [d]_{ker(X)}$ e che tale classe di equivalenza può essere identificata dal valore $X(d)$, uguale per ogni elemento della classe. □

Lemma 6.6. *Siano R_1, R_2 relazioni di equivalenza. Allora, $var(R_1 \cap R_2) \simeq \langle var(R_1), var(R_2)\rangle$.*

Dimostrazione. Abbiamo visto che $ker(var(R_1 \cap R_2)) = R_1 \cap R_2$; inoltre, $(d, d') \in ker(\langle var(R_1), var(R_2)\rangle)$ sse

$$\begin{array}{ccc} \langle var(R_1), var(R_2)\rangle(d) & = & \langle var(R_1), var(R_2)\rangle(d') \\ || & & || \\ \langle var(R_1)(d), var(R_2)(d)\rangle & & \langle var(R_1)(d'), var(R_2)(d')\rangle \\ || & & || \\ \langle [d]_{R_1}, [d]_{R_2}\rangle & & \langle [d']_{R_1}, [d']_{R_2}\rangle \end{array}$$

Per ipotesi abbiamo che $(d, d') \in ker(\langle var(R_1), var(R_2)\rangle)$; quindi, per definizione di kernel e di var, abbiamo che $(d, d') \in R_1$ e $(d, d') \in R_2$, per cui $(d, d') \in R_1 \cap R_2$.
Pertanto, $ker(var(R_1 \cap R_2)) = ker(\langle var(R_1), var(R_2)\rangle)$, e quindi $var(R_1 \cap R_2) \simeq \langle var(R_1), var(R_2)\rangle$. □

Lemma 6.7. *Siano R_1, R_2 relazioni di equivalenza. Allora, $R_1 \subseteq R_2$ implica che $\mathcal{H}(var(R_1)) \geq \mathcal{H}(var(R_2))$.*

Dimostrazione. Se $R_1 \subseteq R_2$ allora $R_1 \cap R_2 = R_1$ e quindi $\mathcal{H}(var(R_1)) = \mathcal{H}(var(R_1 \cap R_2))$. Per il lemma 6.6 e grazie alla Proposizione 6.1, abbiamo che $\mathcal{H}(var(R_1 \cap R_2)) = \mathcal{H}(var(R_1), var(R_2))$. Dato che $\mathcal{H}(X, Y) \geq \mathcal{H}(X)$ possiamo concludere dicendo che

$$\mathcal{H}(var(R_1)) = \mathcal{H}(var(R_1), var(R_2)) \geq \mathcal{H}(var(R_2))$$

□

Lemma 6.8. *Siano R_1, R_2 relazioni di equivalenza e W una variabile aleatoria. Allora, $R_1 \subseteq R_2$ implica che $\mathcal{H}(var(R_1)|W) \geq \mathcal{H}(var(R_2)|W)$.*

Dimostrazione. Come mostrato, $W = var(ker(W))$. Poichè $R_1 \subseteq R_2$, anche $R_1 \cap ker(W) \subseteq R_2 \cap ker(W)$. Applicando il lemma 6.6 abbiamo che

$$var(R_i \cap ker(W)) \simeq \langle var(R_i), var(ker(W))\rangle = \langle var(R_i), W\rangle \quad \text{per } i = 1, 2.$$

Applicando invece il lemma 6.7 otteniamo

$$\mathcal{H}(var(R_1 \cap ker(W))) \geq \mathcal{H}(var(R_2 \cap ker(W)))$$

Per la Proposizione 6.1, ciò implica che

$$\mathcal{H}(var(R_1), W) \geq \mathcal{H}(var(R_2), W).$$

Sottraendo ad ambo i membri la quantità $\mathcal{H}(W)$ e applicando la definizione di entropia condizionata, si ha

$$\begin{array}{ccc} \mathcal{H}(var(R_1), W) - \mathcal{H}(W) & \geq & \mathcal{H}(var(R_2), W) - \mathcal{H}(W) \\ || & & || \\ \mathcal{H}(var(R_1)|W) & & \mathcal{H}(var(R_2)|W) \end{array}$$

□

Consideriamo ora la quantità $\mathcal{I}(Y^{out}; X^{in}|Z^{in})$. Da questa possiamo ottenere l'informazione che passa dagli input bassi del sistema X^{in} agli output Y^{out}, essendo inoltre a conoscenza degli input bassi Z^{in}. Se $\mathcal{I}(Y^{out}; X^{in}|Z^{in}) = 0$ ci aspettiamo che gli input alti non interferiscano con gli output bassi del sistema, e viceversa.

Teorema 6.9. *Sia* $Y^{out} = f \circ < X_1^{in}, \cdots X_n^{in} >$ *un vettore di osservazioni di output. Si ha che* $\{X_1^{in} \cdots X_i^{in}\}$ *non interferisce con* Y^{out} *sse* $\mathcal{I}(Y^{out}; A^{in}|Z^{in})=0$ *dove* $A^{in} =< X_1^{in}, \cdots X_i^{in} >$ *e* $Z^{in} =< X_{i+1}^{in}, \cdots X_n^{in} >$.

Dimostrazione. Per semplicità di notazione chiameremo $A = A^{in}$, $Y = Y^{out}$ e $Z = Z^{in}$. Applicando la definizione di mutua informazione e di entropia condizionata abbiamo che:

$$\mathcal{I}(Y; A|Z) \stackrel{\triangle}{=} \mathcal{H}(Y|Z) + \mathcal{H}(A|Z) - \mathcal{H}(A, Y|Z) \stackrel{\triangle}{=}$$

$$\mathcal{H}(Y|Z) + \mathcal{H}(A, Z) - \mathcal{H}(Z) - \mathcal{H}(A, Y, Z) + \mathcal{H}(Z) = \mathcal{H}(Y|Z) - \mathcal{H}(Y|A, Z)$$

Essendo il sistema deterministico, abbiamo che $Y = f(A, Z)$ e, applicando la proprietà 6.3, otteniamo $\mathcal{H}(Y|A, Z) = 0$. Quindi, per dimostrare il teorema basterà far vedere che A non interferisce con Y sse $\mathcal{H}(Y|Z) = 0$.

(⇒) Per ipotesi abbiamo che A non interferisce con Y; quindi per definizione 6.4.2 $f :=_Z \Rightarrow =_Y$, cioè $\forall(\mu, \mu') \in =_Z .(f(\mu), f(\mu')) \in =_Y$. Ma $(f(\mu), f(\mu')) \in =_Y$ implica che $(\mu, \mu') \in f^{-1}(=_Y)$ e quindi $=_Z \subseteq f^{-1}(=_Y)$.[1] Per il lemma 6.8 otteniamo il seguente risultato:

$$\begin{array}{ccc} \mathcal{H}(var(=_Z)|Z) & \geq & \mathcal{H}(var(f^{-1}(=_Y))|Z) \\ \quad ||^{(1)} & & \quad ||^{(2)} \\ \mathcal{H}(Z|Z) & & \mathcal{H}(Y|Z) \\ \quad || & & \quad \geq \\ \quad 0 & & \quad 0 \end{array}$$

Andiamo ora a dimostrare le uguaglianze (1) e (2).

- **(1)** Basta dimostrare che $=_Z \; = \; ker(Z^{in})$. La coppia $(\mu, \mu') \in ker(Z^{in})$ sse, per definizione di kernel, $Z^{in}(\mu) = Z^{in}(\mu')$, cioè $< X_{i+1}^{in}(\mu), \cdots, X_n^{in}(\mu) >=< X_{i+1}^{in}(\mu'), \cdots, X_n^{in}(\mu') >$. Questa uguaglianza vale sse $X_j^{in}(\mu) = X_j^{in}(\mu')$ per ogni $j = i + 1, .., n$. Ma questo equivale a dire $\forall j = i + 1, .., n \; . \; \mu(x_i) =_Z \mu'(x_i)$, che non è altro che la definizione di $=_Z$.
- **(2)** Basta dimostrare che $f^{-1}(=_Y) = ker(Y^{out})$. La coppia $(\mu, \mu') \in ker(Y^{out})$ sse, per definizione di kernel, $Y^{out}(\mu) = Y^{out}(\mu')$ da cui, banalmente, $Y(\mu) =_Y Y(\mu')$. Poichè $Y = f(A, Z)$, questa

[1] Si noti che, in generale, f non è biunivoca e quindi f^{-1} non è una funzione, ma semplicemente una relazione. Tuttavia, per quello che andremo a dimostrare, questo è del tutto irrilevante.

uguaglianza vale sse $f(< A, Z > (\mu)) =_Y f(< A, Z > (\mu'))$; da ciò, $(< A, Z > (\mu), < A, Z > (\mu')) \in f^{-1}(=_Y)$; visto che $dom(< A, Z >) = \{X_1, ..., X_n\} = dom(\mu) = dom(\mu')$, si ottiene che $< A, Z > (\mu) = \mu$ e $< A, Z > (\mu') = \mu'$, e quindi $(\mu, \mu') \in f^{-1}(=_Y)$.

($\Leftarrow$) L'ipotesi $\mathcal{H}(Y|Z) = 0$ implica, per la proposizione 6.3, che Y è funzione di Z. Per assurdo, supponiamo che A interferisca con Y; quindi, $f :=_Z \not\Rightarrow =_Y$. Allora $\exists \mu, \mu' : (\mu, \mu') \in =_Z$ t.c. $(f(\mu), f(\mu')) \notin =_Y$. Questo implica che deve esistere $y \in Y$ t.c. $f(\mu)(y) \neq f(\mu')(y)$. Sia $y = f(x'_1, \ldots, x'_k)$ per $\{x'_1, \ldots, x'_k\} \subseteq \{x_1, \ldots, x_n\}$. Se ogni $x'_j \in Z$, si avrebbe che $f(\mu)(y) = f(\mu')(y)$, essendo $\mu(x_j) = \mu'(x_j)$ (poiché $\mu =_Z \mu'$ e $x_j \in Z$) e f una funzione; deve quindi esistere j t.c. $x_j \notin Z$ (e quindi $x_j \in X$). Pertanto, Y non è solo funzione di Z: assurdo. □

Parte II

Typed Assembly Language

Capitolo 7

Typed Assembly Language

7.1 Introduzione

Quasi tutti gli utenti che oggi usano del software, si preoccupano soltanto di recuperare i file necessari, di installarli ed eseguirli. Questi passaggi molto semplici, purtroppo, espongono l'utente a seri problemi di sicurezza. In particolare, l'utente, una volta che ha deciso di utilizzare un particolare programma, si *fida (trust)* del codice che sta per utilizzare, quando non vi è alcuna prova che dimostra che questo non sia malizioso. Inoltre, anche se l'utente volesse preventivamente controllare direttamente il codice, in generale, non dispone dei sorgenti. A questo punto, dovrebbe essere chiaro il problema e quanto questo sia attuale. Ciò che vorremmo è uno strumento che, attraverso le specifiche logiche (opportunamente fornite) del software da utilizzare, possa controllarne la correttezza (o parte di questa). Scendiamo un pò di più nei dettagli, cercando di chiarire meglio il problema a basso livello.
In generale, nella creazione di un programma, un compilatore utilizza le informazioni di tipo fornite per creare il codice oggetto. Queste informazioni non vengono però inserite all'interno di quest'ultimo, impedendone (o quasi) il controllo. L'idea è semplice: si vuole mantenere queste informazioni anche nel codice oggetto in modo da facilitarne il controllo attraverso il *type checking*. Anche in questo caso, però, qualcuno potrebbe insinuare che il controllore, il *type checker*, sia malizioso. Tuttavia, è sufficiente considerare che quest'ultimo è un semplice programma, relativamente piccolo e facilmente controllabile direttamente (concetto di *minimal trusted computing base*).
Un *typed assembly language (TAL)* è un linguaggio assembler esteso per includere un metodo per mantenere informazioni di tipo di ogni valore all'interno del codice. Queste informazioni possono essere utilizzate da un programma, il cosidetto *type checker*, che, in sostanza, cerca di analizzare il codice per prevedere come si comporterà una volta eseguito. Più precisamente, tale programma può essere usato per controllare la correttezza del codice, se quest'ultimo rispetta un sistema di tipi appropriato (*type safety*).

Per concludere, TAL non è l'unico strumento per gestire del codice *untrusted*; ve ne sono altri, fra cui il *Proof Carrying Code*. Si tratta di una tecnica particolare, un pò più complessa, secondo la quale (orientativamente): vengono stabilite delle regole che garantiscono il comportamento sicuro dei programmi; chi produce il codice, rispettando le regole stabilite, genera una *safety proof* formale per il codice; chi vuole eseguire il codice può utilizzare un semplice e veloce *proof validator* per controllare che la prova sia valida e che, quindi, il codice è sicuro. Comunque, per quanto interessante, non approfondiremo ulteriormente quest'ultima tecnica.

7.2 Flusso di controllo e tipi di base

In questo paragrafo iniziamo ad introdurre *TAL* formalmente. Verranno fornite le nozioni fondamentali della sintassi, della semantica operazionale e del sistema di tipi. Per semplificare la descrizione, come si vedrà in seguito, si utilizzerà un linguaggio assembler ideale per una macchina astratta semplice.

Definizione 7.2.1.

Sintassi di TAL:

- *Registri* $r ::= r_1 \,|\, r_2 \,|\, ... \,|\, r_n$
- *Labels* $L ::= Identifier$
- *Costanti intere* $n \in [-2^{k-1}, 2^{k-1} - 1]$
- *Blocchi* $B ::= jmp\ v \,|\, i;\ B$
- *Istruzioni* $i ::= aop\ r_d, r_s, v \,|\, bop\ r, v \,|\, mov\ r, v$
- *Operandi* $v ::= r,\ n,\ L$
- *Operazioni Aritmetiche (aop)* $aop ::= add \,|\, sub \,|\, mul \,|\, ...$
- *Operazioni Branch (bop)* $bop ::= beq \,|\, bgt \,|\, ...$

Dalla definizione si può facilmente notare che (ovviamente) ogni blocco termina con jmp *e che, in particolare, è necessario fornire un sistema di tipi, poichè, banalmente, in* jmp v*, l'operando deve essere obbligatoriamente una label.*

Il lettore attento (ricordando le nozioni fondamentali dei corsi di architettura e sistemi operativi) dovrebbe aver notato l'eccessiva semplicità di questa sintassi. Questa, infatti, è sufficiente per poter costruire e descrivere programmi banali, ma non lo è più se volessimo trattare la ricorsione.

A questo punto, dopo aver definito gli strumenti, vediamo come si può descrivere e come si può comportare la nostra macchina astratta ideale. Si tratta, più precisamente, di una macchina a stati la cui funzione di transizione è definita dalla semantica operazionale.

Definizione 7.2.2.

Abstract Machine: $\Sigma = (H, R, B)$

- *H è una funzione parziale che mappa labels in blocchi base B.*
- *R è la funzione che ritorna il valore contenuto all'interno di un registro.*
  - $R(n) = n$
  - $R(L) = L$
  - $R(r_i) = v$ *(nel registro ci può essere un qualsiasi operando)*
- *B è il blocco di base.*

Di seguito viene formalizzata la semantica operazionale:

Regola di inferenza 7.2.1.

- $(H, R, mov\ r_d, v; B) \longmapsto (H, R[r_d := R(v)], B)$
- $(H, R, add\ r_d, r_s, v; B) \longmapsto (H, R[r_d := n], B) \quad se\ n = R(v) + R(r_s)$
- $(H, R, jmp\ v) \longmapsto (H, R, B)$
  $se\ R(v) = L\ e\ H(L) = B$
- $(H, R, beq\ r, v; B) \longmapsto (H, R, B) \quad se\ R(r) \neq 0$
- $(H, R, beq\ r, v; B) \longmapsto (H, R, B')$
  $se\ R(r) = 0, R(v) = L\ e\ H(L) = B'$

La definizione delle altre regole aritmetiche e di salto è lasciata come facile esercizio al lettore, sulla scia di quelle appena presentate.

E' molto importante ricordare che quanto finora descritto è un tipico linguaggio assembler, ovvero non tipato. Infatti, non viene effettuato alcun controllo di tipo. Ad esempio, nell'esecuzione di una somma è necessario essere sicuri di sommare effettivamente due addendi (interi); se ciò non accadesse, ci troveremmo in una situazione di errore nel quale l'evoluzione della macchina sarebbe impossibile. Comunque, è proprio per questo motivo che non trattiamo un linguaggio assembler, bensì un TAL, che garantisce proprio di non entrare mai in uno stato di eroore.

Definizione 7.2.3.

Tipi base:

- $\tau ::= int \mid \Gamma \rightarrow \{\}$
- $\Gamma ::= \{r_1 : \tau_1, r_2 : \tau_2, \ldots\}$

A questo punto dobbiamo vedere come effettuare il type-checking e, quindi, come costruire le nostre regole di tipaggio. Innanzitutto, è ovvio che ad ogni punto del codice dobbiamo tenere traccia sia dei tipi dei registri, sia di quelli delle labels. Per fare ciò definiamo due funzioni particolari (una l'abbiamo appena vista, ma non definita):

Definizione 7.2.4.

- Ψ *: è un mapping da labels a tipi label (*Heap Types*)*
- Γ *: è un mapping da registri a tipi (*Register Types*)*

Altrettanto semplici ed intuitive sono le seguenti regole, necessarie per tipare i vari operandi che, ricordiamo, possono essere: interi, registri, labels. Generalmente si utilizza un giudizio $\Psi; \Gamma \vdash v : \tau$ per tipare gli operandi, mentre $\Psi \vdash B : \Gamma \rightarrow \{\}$ viene utilizzato per tipare i blocchi.

Regola di inferenza 7.2.2.

Typing Operands:

$\Psi; \Gamma \vdash n : int$

$\Psi; \Gamma \vdash r : \Gamma(r)$

$\Psi; \Gamma \vdash L : \Psi(L)$

$$\frac{\Psi; \Gamma \vdash v : \tau_1 \quad \tau_1 \leq \tau_2}{\Psi; \Gamma \vdash v : \tau_2}$$

(subtyping: $\{r_1 : \tau_1, \ldots, r_{i-1} : \tau_{i-1}, r_i : \tau_i\} \leq \{r_1 : \tau_1, \ldots, r_{i-1} : \tau_{i-1}\}$ *)*

Tra queste, l'ultima regola, spesso nota col nome di *Subsumption*, afferma che il programma non andrà mai in crash (ovvero, sarà ben tipato) se tipiamo con un super-tipo. Tale regola, insieme alla definizione di sotto-tipo, non dovrebbe sorprendere poichè, molto probabilmente, è stata già incontrata (forse sotto forme diverse) in linguaggi di programmazione imperativi o a oggetti: ad esempio, un valore intero può essere visto come un reale.
La relazione di sotto-tipo è un ordine parziale, poichè gode delle proprietà riflessiva, antisimmetrica e transitiva. Due aspetti interessanti del subtyping sono le regole di *covarianza* e *controvarianza*.

Definizione 7.2.5.

Covarianza e Controvarianza

$$\frac{A \leq A' \quad B \leq B'}{A \to B \leq A' \to B'} \qquad \frac{A' \leq A \quad B \leq B'}{A \to B \leq A' \to B'}$$

La prima è utilizzata con gli argomenti di funzione, mentre la seconda con i risultati di funzione. Nel nostro caso, i tipi label obbediscono alle regole controvarianti, formalmente:

$$\frac{\Gamma' \leq \Gamma}{\Gamma \to \{\} \leq \Gamma' \to \{\}}$$

Pertanto, possiamo facilmente dedurre che è possibile il subtyping di blocchi; in particolare, per subsumption e controvarianza, abbiamo che

$$\frac{\Psi \vdash B : \Gamma_2 \to \{\} \quad \Gamma_1 \leq \Gamma_2}{\Psi \vdash B : \Gamma_1 \to \{\}}$$

Vediamo ora come possiamo tipare le singole istruzioni. Tipicamente, un giudizio per un'istruzione è del tipo $\Psi \vdash i : \Gamma_1 \to \Gamma_2$. In questo, Γ_1 tiene traccia dei tipi dei registri che sono dati in input all'istruzione, mentre, come si può immaginare, Γ_2 tiene traccia di quelli in output. Le due funzioni possono essere meglio identificate, rispettivamente, come le *typing preconditions* e *typing postconditions*. Per il momento, infine, Ψ è un'invariante, poichè, non avendo alcuno stack a disposizione, non possiamo implementare la ricorsione. Di seguito vengono fornite le regole di tipaggio.

Regola di inferenza 7.2.3.

Typing Istruction:

$$\frac{\Psi;\Gamma \vdash r_s : int \quad \Psi;\Gamma \vdash v : int}{\Psi \vdash aop\ r_d, r_s, v : \Gamma \to \Gamma[r_d := int]} \qquad \frac{\Psi;\Gamma \vdash r : int \quad \Psi;\Gamma \vdash v : \Gamma \to \{\}}{\Psi \vdash bop\ r, v : \Gamma \to \Gamma}$$

$$\frac{\Psi;\Gamma \vdash v : \tau}{\Psi \vdash mov\ r, v : \Gamma \to \Gamma[r := \tau]} \qquad \frac{\Psi;\Gamma \vdash v : \Gamma \to \{\}}{\Psi \vdash jmp\ v : \Gamma \to \{\}}$$

$$\frac{\Psi \vdash i : \Gamma_1 \to \Gamma_2 \quad \Psi \vdash B : \Gamma_2 \to \{\}}{\Psi \vdash i; B : \Gamma_1 \to \{\}}$$

L'ultima regola di riferisce a sequenze di istruzioni (o blocchi). Quella precendente, sulla jump, merita un piccolo approfondimento. Infatti, sebbene il tipaggio possa sembrare particolare, è necessario considerare che a questa istruzione non ne segue un'altra, per la quale il tipaggio della precedente è fondamentale, pertanto il contesto di ritorno è arbitrario. Per semplicità è stato scelto il contesto vuoto, che è il contesto di ritorno per ogni blocco.
Per chiarire meglio i concetti, vediamo un esempio di programma assembler tipato, la funzione fattoriale:

```
fact:   { r1:int, r2:int, r31:{r1:int}-->{} } --> {}
                ble r1, L2
                mul r2, r2, r1
                sub r1, r1, 1
                jmp fact

L2:     { r2:int, r31:{r1:int}-->{} } --> {}
                mov r1, r2
                jmp r31
```

Nell'esempio non sono stati dettagliati i controlli che vengono eseguiti, ma sono banalmente verificabili. Invece, è opportuno che il lettore focalizzi la sua attenzione sulle righe delle labels dei blocchi. Su queste vi è, infatti, specificato il tipo di ogni blocco o, se si preferisce, le precondizioni necessarie per tipare il blocco.

Concludiamo questa parte sulle regole di tipaggio mostrando formalmente come questo avviene sulla nostra macchina, che, ricordiamo, è $\Sigma = (H, R, B)$. In pratica, poichè sappiamo già come tipare un blocco B, ci preoccupiamo di dare una regola di tipaggio per H e R e, infine, per l'intera macchina astratta:

Regola di inferenza 7.2.4.

- *H typing*
$$\frac{Dom(H) = Dom(\Psi) \quad \forall L \in Dom(H) \ t.c. \ \Psi \vdash H(L) : \Psi(L)}{\vdash H : \Psi}$$
- *R typing*
$$\frac{\forall r \in Dom(\Gamma) \ t.c. \ \Psi; \Gamma \vdash R(r) : \Gamma(r)}{\Psi \vdash R : \Gamma}$$
- *Machine typing*
$$\frac{\vdash H : \Psi \quad \Psi \vdash R : \Gamma \quad \Psi; \Gamma \vdash B : \Gamma \to \{\}}{\vdash (H, R, B)}$$

Passiamo ora a formulare la correttezza del sistema di tipi appena presentato. Le dimostrazioni sono lasciate per esercizio al lettore.

Lemma 7.1 (Progress)**.** *Se* $\vdash \Sigma_1$ *(è ben tipato) allora esiste* Σ_2 *tale che* $\Sigma_1 \longmapsto \Sigma_2$ *.*

Dimostrazione. Per induzione sulla struttura del blocco di codice in Σ_1. □

Il lemma *Progress* è necessario per asserire che se un configurazione della macchina è ben tipata allora quella non si trova in uno stato di errore, cioè bloccata. In pratica, la macchina può (giustamente) continuare ad evolvere.

Lemma 7.2 (Subject reduction)**.** *Se* $\vdash \Sigma_1$ *e* $\Sigma_1 \longmapsto \Sigma_2$ *allora* $\vdash \Sigma_2$*.*

Dimostrazione. Per induzione sull'inferenza di $\Sigma_1 \longmapsto \Sigma_2$. □

Tali lemmi sono necessari per la dimostrazione della correttezza del sistema di tipi attraverso il seguente teorema.

Teorema 7.3 (Type Safety)**.** *Se* $\vdash \Sigma$ *e* $\Sigma \to^* \Sigma'$ *allora* Σ' *non è bloccato.*

Dimostrazione. Per induzione sulla lunghezza della computazione. □

Infine, per ricollegarci alla descrizione sommaria di TAL, fornita all'inizio, vediamo due corollari, che derivano dal teorema e dai lemmi precedenti.

Corollario 7.4 (Control-Flow Safety)**.** *Tutte le istruzioni jump sono eseguite verso labels valide.*

Corollario 7.5 (Computation Safety)**.** *Tutti i calcoli aritmetici sono eseguiti con numeri interi.*

Nei prossimi paragrafi vedremo come sarà possibile estendere la nostra macchina astratta e, quindi, il nostro sistema di tipi, in modo tale da avvicinarci sempre di più alla realtà.

7.3 Tipi polimorfi

Aggiungiamo un tipo variabile α che indica un'astrazione sui tipi base; ad esempio:

$$\forall\alpha, \beta\{r_1 : \alpha, r_2 : \beta, r_3 : \{r_1 : \beta, r_2 : \alpha\}\}$$

descrive una funzione che scambia i valori dei registri r_1 e r_2 e inserisce il tipo di ritorno nel registro r_3 per qualsiasi tipo venga assegnato a r_1 e r_2.
Questa modifica ci permette appunto di trattare i tipi in modo astratto e quindi ci consente di riusare il codice dichiarando funzioni parametriche. Oltre a modificare la sintassi dei tipi dobbiamo apportare anche delle modifiche ai salti ad etichette indicando il tipo che assumeranno i tipi polimorfi. Quindi scriveremo *jmp v[τ]* per eseguire un salto ad una funzione polimorfa v impostando il tipo astratto a τ e *jmp* $v[\tau_1 \cdots \tau_n]$ se i tipi polimorfi sono n.
Vediamo ora degli esempi di polimorfismo. In ML possiamo definire la funzione identità nel seguente modo:

```
fun id(n)=n;
```

Il type checker di ML tiperà questa funzione con $\forall\alpha.\alpha \to \alpha$ dove α può assumere qualsiasi tipo. Ad esempio la chiamata $id(3)$ avrà tipo $int \to int$.
Un esempio di una funzione che scambia interi con interi e interi con etichette in TAL potrebbe essere il seguente:

```
swap:
    mov r3,r1
    mov r1,r2
    mov r2,r3
    jmp r31

swapInts:
    jmp swap[int,int]
```

```
swapIntAndLabel:
   mov r31,L
   jmp swap[int,{r2:int}]

L:
   jmp r1
```

Eseguendo il type checking su questo frammento di codice vengono dati i seguenti assegnamenti di tipo: al blocco swap il tipo $\forall\alpha,\beta.\{r_1:\alpha, r_2:\beta, r_{31}:\{r_1:\beta, r_2:\alpha\}\}$, al blocco *swapInts* il tipo $\{r_1:int, r_2:int, r_{31}:\{r_1:int, r_2:int\}\}$, al blocco *swapIntAndLabel* il tipo $\{r_1:int, r_2:\{r_2:int\}\}$ e infine per il blocco etichettato L abbiamo il tipo $\{r_1:\{r_2:int\}, r_2:int\}$.
Un altro esempio di tipi polimorfi in TAL è la chiamata di funzione e la relativa allocazione dei registri. Infatti quando si chiama una funzione, la procedura chiamante memorizza l'indirizzo di ritorno sullo stack e quando termina l'esecuzione della funzione chiamata ne ripristina il contenuto dallo stack ed esegue il salto all'etichetta specificata. Vediamo come viene implementata e tipata l'allocazione dei registri in TAL.

```
caller:
  mov r5,255
  mov r1,5
  mov r31,L
  jmp callee[int]    %{r1:int, r5:int, r31:{r1:int, r5:int}}

callee:      % ∀{r1:int, r5:α, r31:{r1:int, r5:α}}
   mov r4,r5     % salva il contenuto del registro r5
   mov r5,7      % usa il registro r5 per altre operazioni
   add r1,r1,r5
   mov r5,r4     % ripristina il valore del registo r5
   jmp r31        % ritorna al chiamante

L:  % {r1:int, r5:int}
  mul r3,r1,r5
  ...
```

Per provare formalmente la Type Safety della nuova definizione di TAL dobbiamo fare delle modifiche alla semantica operazionale data in precedenza. Abbiamo detto che per noi un programma è una tripla $\Sigma = (H, R, B)$ dove: $H : L \to B$ è una funzione che, data un'etichetta, restituisce il blocco relativo; $R : r \to v$ è una funzione che, dato un nome di un registro, restituisce il valore; e infine B è un blocco. Per trattare i tipi polimorfi bisogna modificare la funzione H in modo tale che, data un'etichetta, restituisca un blocco etichettato, cioè:

$$H(L) = \forall\alpha_1, \cdots, \alpha_n.\Gamma.B$$

dove $\alpha_1, \cdots, \alpha_n$ sono le variabili di tipo che devono essere libere in Γ e in B. Pertanto, la semantica operazionale è definita come segue:

Regola di inferenza 7.3.1.

- $(H, R, mov\ r_d, v; B) \longmapsto (H, R[r_d := R(v)], B)$
- $(H, R, add\ r_d, r_s, v; B) \longmapsto (H, R[r_d := n], B) \quad se\ n = R(v) + R(r_s)$
- $(H, R, jmp\ v[\tau_1, \cdots, \tau_n]) \longmapsto (H, R, B[\tau_1/\alpha_1, \cdots, \tau_n/\alpha_n])$
  $se\ R(v) = L\ e\ H(L) = \forall\alpha_1, \cdots, \alpha_n.\Gamma.B$
- $(H, R, beq\ r, v; B) \longmapsto (H, R, B) \quad se\ R(r) \neq 0$
- $(H, R, beq\ r, v[\tau_1, \cdots, \tau_n]; B) \longmapsto (H, R, B'[\tau_1/\alpha_1, \cdots, \tau_n/\alpha_n])$
  $se\ R(r) = 0, R(v) = L,\ e\ H(L) = \forall\alpha_1, \cdots, \alpha_n.\Gamma.B'$

Dobbiamo ora notare che i tipi possono essere variabili quindi bisogna essere certi che contengano solo variabili correttamente dichiarate.La definizione seguente esprime questo fatto:

Definizione 7.3.1.

Un tipo polimorfo τ si dice ben formato sse:

$$\frac{FreeVars(\tau) \subseteq \Delta \qquad \Delta = \alpha_1, \cdots, \alpha_n}{\Delta \vdash \tau}$$

Presentiamo ora le nuove regole di tipaggio, le restanti rimangono invariate.

Definizione 7.3.2.

- $$\frac{\Psi; \Delta; \Gamma \vdash v : \forall\alpha_1, \alpha_2 \cdots, \alpha_n.\Gamma' \quad \Gamma \vdash \tau}{\Psi; \Delta; \Gamma \vdash v[\tau] : (\forall\alpha_2, \cdots, \alpha_n.\Gamma')[\tau/\alpha_1]}$$
- $$\frac{\begin{array}{l}\forall L \in Dom(H).\Psi; \alpha_1, \cdots, \alpha_n \vdash B : \Gamma \rightarrow \{\} \\ H(L) = \forall\alpha_1, \cdots, \alpha_n.\Gamma.B \\ \Psi(L) = \forall\alpha_1, \cdots, \alpha_n.\Gamma\end{array}}{\vdash H : \Psi}$$

Come per il TAL senza tipi polimorfi, definito nel capitolo precedente, si possono dimostrare per induzione i lemmi di Progress e di Subject reduction.

Lemma 7.6 (Progress)**.** *Se* $\vdash \Sigma_1$ *allora esiste un* Σ_2 *tale che* $\Sigma_1 \longmapsto \Sigma_2$*.*

Dimostrazione. Per induzione sulla struttura di Σ_1. □

Lemma 7.7 (Subject reduction)**.** *Se* $\vdash \Sigma_1$ *e* $\Sigma_1 \longmapsto \Sigma_2$ *allora* $\vdash \Sigma_2$*.*

Dimostrazione. Per induzione sulla lunghezza dell'inferenza $\longmapsto$. □

Tramite i precedenti lemmi possiamo dimostrare il seguente teorema:

Teorema 7.8 (Type Safety)**.** *Se* $\vdash \Sigma$ *e* $\Sigma \longmapsto^* \Sigma'$ *allora* Σ *non si troverà mai in blocco.*

7.4 Stack in TAL

Arricchiamo il nostro modello di macchina astratta aggiungendo una nuova componente: lo *stack*. Difatti lo *stack* è ormai utilizzato in tutti i linguaggi di programmazione in cui è possibile definire funzioni ricorsive; viene infatti sfruttato per memorizzare l'indirizzo di ritorno e le variabili locali prima di una chiamata ricorsiva. Ampliamo, dunque, la definizione 7.2.2 della macchina astratta includendo lo stack:

$$M \triangleq (H, R, S, B)$$

Possiamo vedere lo stack **S** come una lista di valori; formalmente:

$$(Stack) \qquad S ::= nil \mid v :: S$$

dove il termine *nil* viene utilizzato per indicare la fine dello *stack* mentre v rappresenta il valore memorizzato in cima allo *stack*. Riportiamo di seguito le nuove istruzioni che ci permetteranno di agire sullo stack.

$$(Istruzioni) \qquad i ::= salloc\ n \mid sfree\ n \mid sld\ r_d,\ n \mid sst\ r_s,\ n$$

Per tenere traccia della testa dello stack viene riservato un particolare registro che chiameremo *sp* (stack pointer). Le istruzioni *salloc* ed *sfree* vengono utilizzate rispettivamente per allocare e liberare n locazioni sullo stack; ovviamente le operazioni vengono effettuate a partire dalla *testa* della lista, tali operazioni corrispondono dunque a sommare o sottrarre n dallo stack pointer *sp*. La funzione *sld* (*stack load*) memorizza il valore dello stack nella n-esima posizione ($sp(n)$) nel registro di destinazione r_d; infine, *sst* (*stack store*) memorizza nella n-esima posizione dello stack quanto è contenuto nel registro r_s.

Il lettore attento avrà subito notato che troppe deallocazioni, così come letture/scritture in zone non allocate dello stack, portano ad uno stato di errore. Difatti nel sistema dei tipi che analizzeremo tale aspetto verrà affrontato.

Le istruzioni appena introdotte corrispondono alle istruzioni tipiche delle architetture RISC. E' comunque possibile definire istruzioni CISC-like componendo le istruzioni introdotte. Riportiamo due esempi per chiarire il concetto:

$$push\ v\ =\ salloc\ 1;\ \ sst\ v, 1$$

$$pop\ r\ =\ sld\ r, 1;\ \ sfree\ 1$$

Riportiamo a seguire un semplice esempio di programma ricorsivo che calcola il fattoriale. La funzione ricorsiva che studieremo in TAL è:

```
factrec(n) =
                if n ≤ 0
                        then 1
                else
                        n * factrec(n - 1)
```

Nel corrispondente programma TAL l'argomento n viene memorizzato nel registro r_1; il registro r_{31} mantiene l'indirizzo di ritorno che aspetta il risultato in r_1.

```
factrec:
        bgt r1, L1          % se n>0, goto L1
        mov r1, 1
        jmp r31             % se n≤1, return 1

L1:
        salloc 2
        sst r31, 1          % salvo il return address
        sst r1, 2           % salvo n
        sub r1, r1, 1       % n := n - 1
        mov r31, RA         % return address := RA
        jmp factrec         % chiamata ricorsiva

RA:
        sld r2, 2           % restore di n in r2
        sld r31, 1          % restore del return address
        mul r1, r1, r2      % risultato := n * fact(n - 1)
        jmp r31             % ritorno
```

Formalizziamo, così com'è stato fatto in precedenza, la semantica per le istruzioni che riguardano lo stack.

Regola di inferenza 7.4.1.

- $(H, R, S, salloc\ n, B) \longmapsto (H, R, \overbrace{? ::? :: \ldots ::?}^{n} :: S, B)$
- $(H, R, v_1 :: v_2 :: \ldots :: v_n :: S, sfree\ n, B) \longmapsto (H, R, S, B)$
- $(H, R, S, sld\ r, n; B) \longmapsto (H, R[r := v_n], S, B)$
  $dove\ S\ =\ v_1 :: v_2 :: \ldots :: v_n :: S'$
- $(H, R, S_1, sst\ r, n; B) \longmapsto (H, R, S_2, B)$
  $dove\ S_1\ =\ v_1 :: v_2 :: \ldots v_{n-1} :: v_n :: S'\ e$
  $dove\ S_2\ =\ v_1 :: v_2 :: \ldots :: v_{n-1} :: R(r) :: S'$

Dobbiamo fare solo una piccola precisazione sulle regole semantiche introdotte. Ogni sequenza di allocazioni infatti viene effettuata a partire dalla testa dello stack ed il valore di ogni locazione è inizialmente ignoto; per tale motivo, nel nostro modello, abbiamo utilizzato un valore di inizializzazione '*?*'. Va notato che tale valore non viene utilizzato dai programmi ma è utile solo per la semantica operazionale.
Estendiamo l'insieme dei tipi per tipare lo stack.

Definizione 7.4.1.

Tipi per lo stack:

- $\sigma ::= nil \mid \tau :: \sigma \mid \rho$
- $\tau ::= int \mid \Gamma \rightarrow \{\} \mid ?$

Il tipo *nil* è stato introdotto per tipare lo stack vuoto. Diremo, invece, che uno stack $v :: S$ ha tipo $\tau :: \sigma$ quando τ è il tipo di v e σ è il tipo di S. Abbiamo introdotto anche il tipo ρ che descrive una qualche *coda* sconosciuta nello stack. Il tipo ? servirà, come ci si aspetta, per dare un tipo ad ogni un valore di inizializzazione (vd. salloc). Lo stack contiene, come è stato visto in precedenza, un particolare registro che mantiene informazioni sulla locazione della testa dello stack; dal punto di vista dei tipi tale registro avrà come tipo l'intero tipo dello stack:

$$\{sp : int :: \rho, r_1 : int, \ldots\}$$

In aggiunta, è possibile tipare lo stack con un tipo polimorfo:

$$\forall\ \rho.\{sp : int :: \rho, r_1 : int, \ldots\}$$

Mostriamo dunque le regole di tipo. Come in precedenza, le regole hanno la forma:

$$\Psi; \Delta \vdash i : \Gamma_1 \to \Gamma_2$$

Definizione 7.4.2.

$$\frac{\Gamma_1(sp) = \sigma}{\Psi; \Delta \vdash salloc\ n : \Gamma_1 \to \Gamma_1[sp :=? ::? :: \ldots ::? :: \sigma]}$$

$$\frac{\Gamma_1(sp) = \tau_1 :: \tau_2 :: \ldots :: \tau_n :: \sigma}{\Psi; \Delta \vdash sfree\ n : \Gamma_1 \to \Gamma_1[sp := \sigma]}$$

$$\frac{\Gamma(sp) = \tau_1 :: \tau_2 :: \ldots :: \tau_n :: \sigma}{\Psi; \Delta \vdash sld\ r, n : \Gamma \to \Gamma[r := \tau_n]}$$

$$\frac{\Gamma(sp) = \tau_1 :: \tau_2 :: \ldots :: \tau_n :: \sigma}{\Psi; \Delta \vdash sst\ r, n : \Gamma \to \Gamma[sp := \tau_1 :: \tau_2 :: \ldots :: \tau_{n-1} :: \Gamma(r) :: \sigma]}$$

Capitolo 8

Compilazione in TAL

Finora sono state mostrate le funzionalità di cui il linguaggio **TAL** dispone: la possibilità di implementare un meccanismo di controllo dei tipi (eventualmente polimorfico) in un linguaggio di basso livello ci permette di poter effettuare dei controlli sul codice oggetto e di scartare programmi potenzialmente dannosi (ad esempio impedendo ad un programma di accedere a zone di memoria protetta). L'uso del linguaggio **TAL** ci permette dunque di ottenere uno strumento di controllo e verifica dei programmi che risulta estremamente utile.
Tuttavia scrivere un programma direttamente in assembly tipato richiederebbe la scrittura a mano di una grande quantità di codice, di cui deve in ogni caso essere verificata la correttezza. Chiaramente, cercare di trovare un errore che porta al malfunzionamento di un programma in mezzo a milioni di linee di codice assembly non è praticabile, senza contare il fatto che tipicamente quelle che si devono controllare sono istruzioni di basso livello.
La soluzione più conveniente è quella di definire un linguaggio di programmazione di alto livello e, per ognuno dei costrutti sintattici ammessi da tale linguaggio, specificare una regola di conversione di tale costrutto in una sequenza di istruzioni **TAL**; al meccanismo di compilazione che andremo a realizzare dovrà però essere dedicata parecchia attenzione. Difatti ciò che si vuole fare è partire da un linguaggio di alto livello ed effettuarne una traduzione in **TAL**. Si noti l'assunzione che anche il linguaggio sorgente sia un linguaggio tipato. Vogliamo allora essere sicuri che, ogni qualvolta un programma scritto nel linguaggio sorgente risulta tipabile, allora anche la rispettiva traduzione in **TAL** dovrà essere tipabile.
In conclusione, le componenti di cui abbiamo bisogno per effettuare la compilazione sono le seguenti:

- Un linguaggio sorgente tipato.
- Un linguaggio di destinazione (**TAL** nel nostro caso), anch'esso tipato.
- Un compilatore che preservi i tipi, vale a dire programmi tipabili vengono tradotti in programmi a loro volta tipabili.

8.1 Il linguaggio Tiny

Quello che svilupperemo, per mostrare il meccanismo di compilazione, è un linguaggio minimale che possegga però tutte le caratteristiche principali tipiche di un linguaggio di programmazione, quali la definizione di funzioni, la valutazione di espressioni e il test di equivalenza per 0.
Di seguito indicheremo con $f, g, \cdots$ gli identificatori di funzione, mentre con $x, y, \cdots$ gli identificatori di variabili. Il linguaggio **Tiny** che andremo a presentare è un linguaggio di tipo funzionale, in cui un'espressione può essere un identificativo (di variabile o di funzione), oppure è dato dall'applicazione di una qualche operazione su più espressioni (ad esempio la somma di espressioni), un costrutto condizionale e una dichiarazione locale.

Definizione 8.1.1.

Espressioni nel linguaggio ***Tiny****:*

$$e ::= \ x \mid f \mid e_1 + e_2 \mid e_1 e_2 \mid if\ e_1 = 0\ then\ e_2\ else\ e_3 \mid let\ x = e_1\ in\ e_2$$

L'unico tipo di base di cui si dispone nel linguaggio **Tiny** è il tipo *int*. Tuttavia, una funzione che, ad esempio, prende in input un intero e restituisce un intero, avrà tipo $int \mapsto int$. Inoltre una funzione che prende in input più argomenti, di tipo $\tau_1, \cdots, \tau_n$ può essere vista come una funzione che prende in input un solo argomento di tipo $\tau_1 \times \cdots \times \tau_n$.

Definizione 8.1.2.

Tipi presenti nel linguaggio ***Tiny****:*

$$\tau ::= int \mid \tau_1 \times \tau_2 \mid \tau_1 \mapsto \tau_2$$

Le funzioni in Tiny prenderanno come argomento un solo parametro e restituiranno un valore di un certo tipo. Se τ_1 è il tipo del parametro richiesto in ingresso (x_1), τ_2 è il tipo dell'oggetto restituito dalla funzione ed e è il corpo della funzione, allora useremo la seguente notazione:

$$f(x_1 : \tau_1) : \tau_2 = e$$

E' ora possibile definire la struttura sintattica di un programma **Tiny**, che consisterà di una serie di dichiarazioni di funzioni e nella valutazione di un'espressione.

Definizione 8.1.3.

Siano $d_1, d_2, \cdots, d_n$ *delle funzioni; allora*

$$\begin{array}{rll} P ::= & let & \\ & & d_1, d_2, \cdots, d_n \\ & in & \\ & & e \end{array}$$

è un programma sintatticamente corretto in ***Tiny****.*

La valutazione dei tipi avverrà in base ad una funzione parziale ϕ, che associa agli identificatori un tipo. In un contesto ϕ potremo dire allora che, se la funzione è definita sull'identificatore di variabile x, allora la variabile x ha tipo $\phi(x)$. Poiché si presume che a questo punto il lettore abbia una buona familiarità con i sistemi di tipi, forniamo le regole di inferenza per il sistema di tipi di **Tiny** senza scendere nel dettaglio.

Regola di inferenza 8.1.1.

Sistema di tipi di ***Tiny****:*

$$\frac{}{\phi \vdash x : \phi(x)} \qquad \frac{}{\phi \vdash f : \phi(f)}$$

$$\frac{}{\phi \vdash n : int}$$

$$\frac{\phi \vdash e_1 : int \quad \phi \vdash e_2 : int}{\phi \vdash e_1 + e_2 : int} \qquad \frac{\phi \vdash e_1 : \tau_1 \mapsto \tau_2 \quad \phi \vdash e_1 : \tau_1}{\phi \vdash e_1 e_2 : \tau_2}$$

$$\frac{\phi \vdash x : int \quad \phi \vdash e_1 : \tau \quad \phi \vdash e_2 : \tau}{\phi \vdash \textit{if } x = 0 \textit{ } e_1 \textit{ else } e_2 : \tau} \qquad \frac{\phi \vdash e_1 : \tau_1 \quad \phi, x : \tau_1 \vdash e_2 : \tau_2}{\phi \vdash \textit{ let } x = e_1 \textit{ in } e_2 : \tau_2}$$

$$\frac{\phi, x : \tau_1 \vdash e : \tau_2}{\phi \vdash [f(x : \tau_1) : \tau_2 = e] : (f : \tau_1 \mapsto \tau_2)}$$

Sia allora $\phi : [f_1 : \tau_{1,1} \mapsto \tau_{1,2}, \cdots, f_n : \tau_{n,1} \mapsto \tau_{n,2}]$*, avremo la seguente regola di tipaggio per un programma*

$$\frac{\phi \vdash d_i : (f_i : \tau_{i,1} \mapsto \tau_{i,2}) \quad \phi \vdash e : int}{\vdash \textit{letrec } d_1, \cdots, d_n \textit{ in } e : int}$$

8.2 Conservazione dei tipi

Un compilatore può essere visto come un algoritmo che, partendo da un programma scritto in un linguaggio sorgente, effettua una serie di trasformazioni prima di ottenere un programma semanticamente equivalente in un linguaggio di destinazione. Anzitutto, ognuna delle trasformazioni che avvengono durante il processo di compilazione deve conservare i tipi. Da questo punto di vista una trasformazione, all'interno della compilazione, può essere divisa in due parti:

- Una traduzione dei tipi del linguaggio sorgente nei tipi del linguaggio di destinazione.
- Una traduzione dei termini del linguaggio sorgente nei termini del linguaggio di destinazione.

In merito alla traduzione di Tiny in **TAL**, ci chiediamo come debbano essere tradotti i tipi cosicché vengano preservati dal compilatore. Indubbiamente, il tipo di dato intero *int* non ha bisogno di essere tradotto, poiché disponiamo

dello stesso tipo in **TAL**. Per indicare la traduzione di un tipo τ useremo la notazione $\mathscr{T}[\![\tau]\!]$; avremo allora che

$$\mathscr{T}[\![int]\!] = int$$

Maggiore attenzione andrà invece dedicata al tipo di dato $\tau_1 \mapsto \tau_2$: ovviamente ci aspettiamo che la traduzione dei tipi sia ricorsiva. Inoltre, per poter descrivere come si comporta un tipo appartenente a tale categoria, bisognerebbe conoscere come viene tradotta una chiamata a funzione; in effetti tale tipo può essere associato solamente ad una funzione che prende in ingresso un argomento di tipo τ_1 e restituisce un valore di tipo τ_2. Il meccanismo di traduzione che utilizzeremo per le funzioni sarà quello classico, in cui il valore dell'argomento e l'indirizzo di ritorno della funzione vengono inseriti in cima allo stack prima di procedere ad eseguire il corpo della funzione, mentre al termine delle istruzioni contenute in essa verranno ripristinati i due valori in cima allo stack con delle operazioni di pop, e il risultato della funzione viene memorizzato nel registro r_a.
Alla luce di queste osservazioni, utilizzeremo la notazione

$$\mathscr{K}[\![\tau, \sigma]\!] \stackrel{\triangle}{=} \{sp : \sigma, r_a : \mathscr{T}[\![\tau]\!]\}$$

per indicare il tipo di una label in cui lo stack ha tipo σ e il registro r_a ha per tipo la traduzione del tipo τ. Tenendo conto della politica che verrà adottata per la traduzione delle funzioni, si ha che al momento di effettuare il salto all'inizio della funzione, nello stack saranno presenti una label che si riferisce all'indirizzo di ritorno della funzione, a seguire il valore dell'argomento passato alla funzione, e infine ciò che vi era nello stack prima di eseguire la chiamata a funzione. Capire quale sia il tipo della label in cima allo stack risulta semplice, poiché si ricorda che verrà effettuata l'estrazione degli elementi inseriti nello stack prima di iniziare ad eseguire il corpo della funzione; pertanto, lo stack pointer *sp* dovrà avere lo stesso tipo di quello che vi era nello stack precedentemente alla chiamata di funzione. Inoltre, supponendo che la funzione che stiamo analizzando abbia tipo $\tau_1 \mapsto \tau_2$, nel registro r_a dovrà essere inserito un valore di tipo τ_2: avremo allora che il tipo della label che si riferisce al blocco in cui si dovrà continuare a eseguire il codice una volta usciti dalla funzione è $\mathscr{K}[\![\tau_2, \rho]\!]$. Se si considera il contenuto dello stack all'inizio dell'esecuzione del corpo della funzione, si ha che la traduzione di tipo

$$\mathscr{T}[\![\tau_1 \mapsto \tau_2]\!] = \forall\rho\{sp : \mathscr{K}[\![\tau_2, \rho]\!] :: \mathscr{T}[\![\tau_1]\!] :: \rho\}$$

dove viene implicitamente specificato che lo stack (o meglio il tipo dello stack pointer) resta immutato al termine della chiamata a funzione[1].

8.3 Traduzione di espressioni

Si è già visto, sebbene in maniera informale, come si possa utilizzare lo stack per la traduzione delle funzioni. Per ottenere un compilatore che operi in ma-

[1] Si noti infatti che viene utilizzato lo stesso tipo ρ in $\mathscr{K}[\![\cdot]\!]$ e in fondo allo stack.

niera abbastanza semplice, ricorreremo a una filosofia di pensiero simile anche per la valutazione delle espressioni, utilizzando dunque in maniera massiccia le operazioni a disposizione sullo stack.
L'idea alla base del processo di traduzione di un'espressione è infatti quello di mantenere, all'interno dello stack, i valori di tutte le variabili coinvolte nella valutazione dell'espressione stessa. Anche in questo caso si dovrà procedere in maniera ricorsiva, vale a dire per poter valutare un'espressione sarà necessario dapprima valutare le sottoespressioni di cui è composta. Così come per le funzioni, anche per le espressioni il risultato finale verrà memorizzato nel registro r_a. La traduzione dovrà essere effettuata in maniera tale che i tipi vengano preservati rispetto alla traduzione dei tipi illustrata nella sezione precedente.
Per comodità di notazione:

- La traduzione di un'espressione viene indicata con la notazione $\mathscr{E}[\![\cdot]\!]_{M,\sigma}$, dove M e σ sono dei parametri che indicano rispettivamente una funzione che associa ad ogni variabile un offset all'interno dello stack e un tipo per lo stesso.
- Supporremo che ad ogni funzione f sia associata una label L_f.
- Indicheremo con $\mathcal{T}(e)$ il tipo sorgente dell'espressione e.

I casi più semplici si hanno quando si deve valutare una costante, oppure una funzione; i due casi sono simili, e si ha:

$$\begin{aligned} \mathscr{E}[\![n]\!]_{M,\sigma} &= \text{mov } r_a, n \\ \mathscr{E}[\![f]\!]_{M,\sigma} &= \text{mov } r_a, L_f \end{aligned}$$

Nel caso in cui si ha una variabile le cose sono leggermente più complicate; ogni variabile viene difatti memorizzata all'interno dello stack, in una posizione indicata dalla funzione M. Abbiamo la seguente traduzione:

$$\mathscr{E}[\![x]\!]_{M,\sigma} = \text{sld } r_a, M(x)$$

Definiti i casi base, rimane la valutazione di espressioni composte da più espressioni: in particolare studieremo il caso della somma di espressioni, della composizione e del controllo del flusso di esecuzione.
Se vogliamo valutare un'espressione del tipo $e_1 + e_2$, si dovrà dapprima valutare e_1; quindi effettuare il push del risultato (posto in r_a) in cima allo stack; valutare e_2; e infine si recuperare il risultato della valutazione di e_1 dalla cima dello stack, e si procedere alla somma delle due valutazioni. Tuttavia, si deve prestare attenzione al fatto che gli offset delle variabili, una volta effettuato il push del valore ottenuto dalla valutazione di e_1, dovranno essere incrementati di 1. Dovremo quindi utilizzare una nuova funzione M' tale che:

$$M'(x) = M(x) + 1 \quad \forall x \in dom(M)$$

Definiamo la funzione I in modo che $I(M) = M'$. Siamo allora in grado di dare la traduzione della somma di espressioni:

$$\begin{aligned}\mathscr{E}[\![e_1 + e_2]\!]_{M,\sigma} = \\ &\mathscr{E}[\![e_1]\!]_{M,\sigma} \\ &\text{push } r_a \\ &\mathscr{E}[\![e_2]\!]_{I(M),int::\sigma} \\ &\text{pop } r_t \\ &\text{add } r_a, r_t, r_a\end{aligned}$$

dove r_t è un registro che verrà utilizzato solamente quando si vorrà caricare il valore calcolato da una sottoespressione (non vi è quindi il rischio di andare a sovrascrivere un valore che si vuole mantenere in memoria).
Il lettore, dopo aver compreso il modo in cui si opera per fornire la traduzione delle espressioni, può verificare che le seguenti regole di traduzione sono corrette:

$$\begin{aligned}\mathscr{E}[\![e_1 e_2]\!]_{M,\sigma} = \\ &\mathscr{E}[\![e_1]\!]_{M,\sigma} \\ &\text{push } r_a \\ &\mathscr{E}[\![e_2]\!]_{I(M),\mathscr{T}[\![\tau_1]\!]::\sigma} \\ &\text{pop } r_t \\ &\text{push } L_r[\rho] \\ &\text{jmp } r_t[\sigma] \\ \\ &L_r : \forall\rho\mathscr{K}[\![\tau_2, \sigma]\!]\end{aligned}$$

dove $\mathcal{T}(e_1) = \tau_1 \mapsto \tau_2$ e L_r è una label nuova. Per il controllo del flusso di esecuzione si ha invece:

$$\begin{aligned}\mathscr{E}[\![\text{if } e_1 = 0 \text{ then } e_2 \text{ else } e_3]\!]_{M,\sigma} = \\ &\mathscr{E}[\![e_1]\!]_{M,\sigma} \\ &\text{bneq } r_a, L_{else}[\rho] \\ &\mathscr{E}[\![e_2]\!]_{M,\sigma} \\ &\text{jmp } L_{end}[\rho] \\ \\ &L_{else} : \forall\rho\{sp : \sigma\} \\ &\mathscr{E}[\![e_3]\!]_{M,\sigma} \\ &\text{jmp } L_{end}[\rho] \\ \\ &L_{end} : \forall\rho\mathscr{K}[\![\tau, \sigma]\!]\end{aligned}$$

8.4 Traduzione di programmi

L'ultimo passo da effettuare per completare il lavoro di traduzione di un programma è quello di considerare la definizione di funzioni, oltre che il programma stesso.

Per quanto riguarda la definizione delle funzioni, il ragionamento è piuttosto semplice; in prima istanza bisogna dichiarare la label L_f associata alla funzione (in maniera tale che, quando viene effettuata una chiamata a tale funzione nel programma principale, si cominci effettivamente ad eseguire il corpo di tale funzione con l'argomento necessario), e poi si valuterà l'espressione rispetto all'argomento passato. Si ricordi che, per come è stata definita la traduzione della chiamata a funzione, allora il valore del parametro passato come argomento sarà nella seconda posizione a partire dalla cima dello stack, mentre in testa avremo l'indirizzo di ritorno. Pertanto eseguire il corpo della funzione di tipo $\tau_1 \mapsto \tau_2$ corrisponderà a valutare l'espressione ad essa associata nell'ambiente in cui $dom(M) = \{x\}, M(x) = 2$, e nella quale lo stack avrà un oggetto di tipo $\mathscr{K}[\![\tau_2, \rho]\!]$ in testa, un oggetto del tipo $\mathscr{T}[\![\tau_1]\!]$ richiesto come input della funzione, e a seguire il resto dello stack. La traduzione risultante è la seguente:

$$\begin{aligned}\mathscr{F}[\![f(x:\tau_1):\tau_2 = e]\!] = \\ & L_f : \mathscr{T}[\![\tau_1 \mapsto \tau_2]\!] \\ & \mathscr{E}[\![e]\!]_{[x:=2],\mathscr{K}[\![\tau_2,\rho]\!]::\mathscr{T}[\![\tau_1]\!]::\rho} \\ & \text{pop } r_t \\ & \text{sfree 1} \\ & \text{jmp } r_t\end{aligned}$$

L'ultimo passo necessario per terminare il lungo lavoro svolto è quello di fornire la regola di traduzione dei programmi. Anche questa, disponendo oramai degli strumenti necessari per tradurre la definizione di funzioni e la valutazione di espressioni, risulta facile:

$$\begin{aligned}\mathscr{P}[\![\text{letrec } d_1, d_2, \cdots, d_n \text{ in } e]\!] = \\ & \mathscr{F}[\![d_1]\!] \\ & \mathscr{F}[\![d_2]\!] \\ & \cdots \\ & \mathscr{F}[\![d_n]\!] \\ & L_{main} : \forall\rho\{sp : \mathscr{K}[\![int, \rho]\!] :: \rho\} \\ & \mathscr{E}[\![e]\!]_{\cdot,\mathscr{K}[\![int,\rho]\!]::\rho} \\ & \text{pop } r_t \\ & \text{jmp } r_t\end{aligned}$$

8.5 Conclusioni

Il processo di traduzione mostrato mette in evidenza come sia possibile, dato un linguaggio di programmazione tipato come **Tiny**, effettuarne la compilazione all'interno di **TAL**. Sebbene non viene mostrato formalmente, tale processo di compilazione garantisce che un programma ben tipato in **Tiny** venga tradotto in un programma che sia a sua volta ben tipato in **TAL**. Tale operazione è stata effettuata utilizzando un numero di registri estremamente basso, mentre si è

utilizzato ampiamente lo stack per eseguire la maggior parte delle operazioni. Quello che si è voluto provare infatti era la possibilità di costruire un compilatore inerente alle specifiche richieste (nel nostro caso la conservazione dei tipi), ma non si è considerato l'aspetto dell'efficienza. Difatti, nella quasi totalità dei casi il codice prodotto risulta estremamente ridondante: si pensi alla somma di due espressioni; ciò che si fa in genere è valutare la prima espressione, inserire il risultato in cima allo stack, valutare la seconda espressione, recuperare il risultato della prima sottoespressione e procedere infine ad effettuare la somma. In realtà sarebbe molto più comodo memorizzare direttamente i risultati delle due sottoespressioni in due registri diversi, e procedere alla somma dopo aver eseguito un numero di istruzioni estremamente ridotto. Un meccanismo di ottimizzazione del codice che massimizzi il numero di registri utilizzati[2], minimizzando in tal modo il numero di operazioni necessarie sullo stack. Tuttavia, tali meccanismi esulano dallo scopo del corso e non verranno trattati.

[2]Tale procedimento acquisterebbe valore pratico assumendo che il numero di registri a nostra disposizione sia limitato

Parte III

Reference Monitor

Capitolo 9

Reference Monitor

9.1 Introduzione

Nell'architettura dei sistemi operativi ordierni, un *reference monitor* è un modulo che viene sempre invocato per garantire il rispetto di una o più politiche di sicurezza. Poiché la funzione che svolge è particolarmente rilevante, il modulo deve essere sicuro, o quantomeno difficile da compromettere. E' necessario precisare che il pericolo può non derivare semplicemente da un intervento malizioso esterno al sistema. Limitarsi ad una tale assunzione rappresenta un'estrema debolezza, perché è molto facile incorrere in errori di programmazione che, verificandosi, renderebbero il sistema vulnerabile. Per questo motivo, il "controllore" viene costruito seguendo, come al solito, il principio di *minimal trusted computing base*. Si cerca, in pratica, di mantenerlo quanto più semplice e piccolo possibile in modo tale da poterlo analizzare e testare completamente. Il reference monitor, come appena accennato, controlla l'esecuzione dei processi, o magari dell'intero sistema. In particolare, osserva ma non prevede, ovvero: nota la violazione di una politica durante il flusso di esecuzione, ma non la prevede. Pertanto, non è possibile ottenere proprietà di non-interference. Il rpoblema principale dei reference monitors è il costo computazionale, tutt'altro che irrilevante. Gli estremi che potremmo avere sono: il reference monitor blocca ogni cosa e la complessità è minima (se non nulla); oppure, il reference monitor, molto complesso, riesce ad osservare e comprendere ogni evento, bloccando tutte e sole le situazioni in cui una data politica di sicurezza viene violata. Ovviamente, ciò che vogliamo è un accettabile equilibrio fra i due casi. Inoltre, è importante considerare che il concetto di Reference Monitor è molto generale: potrebbe essere implementato come un processo separato, oppure potrebbe essere inglobato all'interno del sistema da monitorare (*Inlined Reference Monitor*), evitando così il process switch.
Negli argomenti che ci accingiamo a trattare vedremo come formalizzare il concetto di reference monitor e ne studieremo le classi di complessità.

9.2 Politiche di Sicurezza

Una *Politica di Sicurezza* indica un insieme di esecuzioni (di passi) che non sono accettabili all'interno del sistema. Molte di queste sono certamente già note, mentre qualche altra è stata appresa nei capitoli precedenti. Per fare degli esempi, queste possono riguardare: il controllo degli accessi, i flussi di informazione e molto altro. Le politiche di sicurezza possono essere focalizzate meglio se vengono associate al *principio del privilegio minimo*: ogni modulo computazionale (processo, programma, utente, ...) deve avere visibilità delle sole risorse e informazioni immediatamente necessarie al suo funzionamento.
Iniziamo, dunque, col fare una panoramica della situazione. Abbiamo una politica di sicurezza che vogliamo far rispettare in un determinato sistema (nel senso generale del termine), che chiameremo *target*. Per raggiungere il nostro obiettivo ci serviamo di un "processo", parallelo al target, che ne controlla le azioni (monitoring) e, se queste infrangono le specifiche della politica, le blocca. In generale, ciò viene chiamato *EM (Execution Monitoring) enforcement mechanism*.

Definizione 9.2.1.

- S = *target*
- Σ_S = *esecuzioni del target*
- P = *politica di sicurezza (predicato su insiemi di esecuzioni)*

Per quanto definito, possiamo affermare che: un target S soddisfa una politica di sicurezza P se e solo se $P(\Sigma_S) = true$.
Vediamo alcune importanti ed intuitive proprietà riguardanti la classe EM ($\hat{P}$ è un predicato sulle esecuzioni; σ è un'esecuzione).

Proprietà 9.1 (EM)**.**

EM1: $\exists \hat{P}$ *t.c.* $P(\Sigma_S) \iff \forall \sigma \in \Sigma_S \;\; \hat{P}(\sigma)$

EM2: $\hat{P}(\sigma) \implies \forall \sigma' \leq \sigma \;\; \hat{P}(\sigma')$

EM3: $\neg \hat{P}(\sigma) \implies \exists \sigma' \leq_{fin} \sigma \;\; \neg \hat{P}(\sigma')$

EM4: $\hat{P}$ *è decidibile su esecuzioni finite*

La prima riguarda la definizione di politica di sicurezza: essa deve essere definita sulle singole tracce e non sul comportamento globale del target (questo rende, per esempio, proprietà di non-interferenza non ottenibili tramite EM). La seconda è la proprietà di chiusura per prefissi: se la politica è rispettata da una

certa esecuzione allora è necessariamente rispettata in tutti i passi precedenti. La terza afferma che una esecuzione che non rispetta la politica viene bloccata dopo un periodo finito. Infine, la quarta indica la realizzabilità concreta (tramite una macchina di Turing) che, in un numero finito di passi, si fermerà e accetterà/rifiuterà.
Per finire, delineiamo meglio il quadro di azione di un monitor fornendo esplicitamente alcune osservazioni:

Osservazione 9.2.

- *Un monitor realizzato per garantire il rispetto di una politica di sicurezza non deve essere aggirabile.*
- *Politiche che pongono vincoli sul futuro non possono essere gestite con monitor (anche nell'introduzione abbiamo accennato che questi osservano ma non prevedono).*
- *Un monitor non può garantire totale assenza di starvation.*

9.2.1 Automi e Sicurezza

Precedentemente abbiamo affermato che una esecuzione rispetta una politica di sicurezza P se il predicato $\hat{P}$ è vero per ogni passo (prefisso). Definiamo ora un modello formale che, per ogni avanzamento del sistema, controlla il predicato relativo all'azione avvenuta, aggiornandosi per poter controllare l'azione successiva. Tale modello prende il nome di *Security Automata*. Dal punto di vista concettuale, possiamo assumere che il sistema parta da uno stato consistente. Successivamente, poichè ogni azione comporta una evoluzione del sistema, il nostro automa si preoccuperà di effettuare i relativi controlli e, se questi avranno esito positivo, transiterà in un nuovo stato, che rappresenta la configurazione del sistema in seguito all'azione.

Definizione 9.2.2.

Un Security Automata è una quadrupla $(Q,\ Q_0,\ I,\ \rho)$ *con le seguenti caratteristiche*

- Q*: insieme numerabile di stati*
- Q_0*: insieme numerabile di stati iniziali,* $Q_0 \subseteq Q$
- I*: insieme numerabile di simboli di input*
- ρ*: è la funzione di transizione* $Q \times I \rightarrow 2^Q$ *(insieme delle parti), definita induttivamente come:*

$$\begin{aligned} \rho(Q_0, \epsilon) &= Q_0 \\ \rho(Q_0, s_1, \ldots, s_n, s_{n+1}) &= \bigcup_{q \in \rho(Q_0, s_1, \ldots, s_n)} \rho(q, s_{n+1}) \end{aligned}$$

L'automa presentato è molto simile ad un generico automa non-deterministico; le due peculiarità sono l'assenza di stati finali e l'aver definito ogni insieme come numerabile, quindi potenzialmente infinito. Questo non dovrebbe stupire perchè non possiamo porre limiti sull'attività di un sistema.
Per fare un esempio, basta pensare ad un server: in generale, lo scopo di questo è fornire sempre un servizio ad un numero di richieste illimitato nel tempo. I simboli di input rappresentano un insieme che può essere formato da stati del sistema, azioni atomiche, ... Poichè si è abituati a pensare ad un automa come una macchina che accetta o rifiuta, possiamo definire una condizione di accettazione per un sequenza come l'esistenza di una serie di transizioni che portano in un sottoinsieme *non vuoto* di Q. Formalmente, l'insieme delle sequenze accettate dall'automa è:

$$L \triangleq \{s_1, s_2, \ldots \mid \forall\, i \;\; \rho(Q_0, s_1, \ldots, s_i) \neq \emptyset\}$$

Per fare un esempio, di seguito è presentato un esempio di Security Automata per l'enforcement di una politica di sicurezza che proibisce l'esecuzione di una Send dopo che una FileRead è stata eseguita (nfr: no file read; fr: file read).
Vediamo di comprendere meglio come avviene una di queste transizioni. Definiamo p_{ij} come il predicato che rappresenta l'etichetta dell'arco tra il nodo i e il nodo j. Questo ha valore vero se e solo se il simbolo di input soddisfa il predicato. Il nostro automa, supponendo che si trovi in uno stato q', in seguito alla lettura di un simbolo di input s (un valore della sequenza), effettuerà:

$$(q', s) \rightarrow \{q_j \mid q_i \in Q' \wedge p_{ij}(s)\}$$

Nonostante la loro semplicità, è facile notare come questi automi possiedano un grado di impraticabilità. Infatti, se l'insieme degli stati fosse molto grande o la funzione di transizione fosse abbastanza complessa, essi sarebbero alquanto difficili da descrivere. Introduciamo, quindi, un nuovo e semplice formalismo, i *guarded commands*, $B \rightarrow S$. B è chiamato *guard* ed S è chiamato *command*.

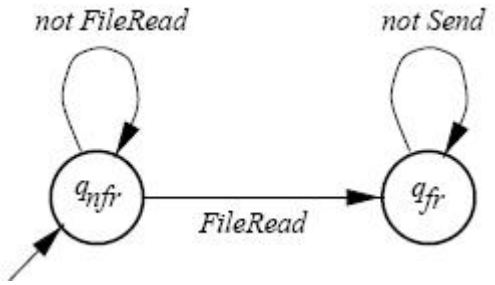

Figura 9.1: No Send after FileRead

In pratica, viene asserito che: la transizione di stato definita da un'azione S si verifica ogni volta che il predicato B è soddisfatto dal simbolo di input e dallo stato corrente. Vediamo come potremmo tradurre l'esempio precedente:

```
state:
   {0, 1}, initial 0 (codifica degli stati)

transitions:
   (NOT FileRead) AND (state == 0)  -->  skip
   (FileRead) AND (state == 0)  -->  state = 1
   (NOT Send) AND (state == 1)  -->  skip
   default  -->  REJECT
```

Si può immaginare come una 'esplosione' degli stati può essere meglio tollerata e, soprattutto, come questi possono essere più facilmente descritti.
Questo semplice esempio conclude la nostra trattazione. Nei prossimi argomenti vedremo lo sviluppo di nuove tecniche ed una utile panoramica sulle classi di complessità relative ai Reference Monitor.

9.3 Program Machine

Modelliamo il concetto di programma da osservare ricorrendo alle macchine di Turing. Ciò ci permetterà di ottenere un riferimento unico per tutte le tecniche utilizzate (Analisi Statica, Execution Monitoring, ...). Considerando il fatto che un programma riceve in ingresso dei dati di input e restituisce in output un risultato, utilizzeremo una macchina di Turing non deterministica a tre nastri: Input(sola lettura), Output (sola scrittura) e Lavoro (lettura e scrittura).

Definizione 9.3.1.

Una Program Machine è una quintupla:

$$PM = (Q, q_0, \Gamma, \llcorner, \delta)$$

dove Q è un insieme di stati, q_0 denota lo stato iniziale, Γ denota l'alfabeto di input, $\llcorner \in \Gamma$ è il blank symbol *e δ è la funzione di transizione:*

$$\begin{aligned} \delta : Q \times \Gamma \rightarrow & \quad Halt \\ \uplus & \quad Q \times \{-1, 0, 1\} \\ \uplus & \quad Q \times \Gamma \times \{-1, 0, 1\} \\ \uplus & \quad \Gamma \times \{-1, 0, 1\} \times Q \times \{-1, 0, 1\} \\ \uplus & \quad \Gamma \rightarrow \cdots \end{aligned}$$

Intuitivamente, la funzione di transizione specifica quali azioni possono possono essere intraprese in uno stato q, leggendo un simbolo γ:

- la Program Machine si ferma (questo caso viene gestito esplicitamente utilizzando il simbolo $Halt$);
- si cambia stato e si muove la testina di input;
- viene cambiato stato, si scrive sul nastro di lavoro e viene mossa la testina del nastro di lavoro;
- la Program Machine scrive sul nastro di lavoro, cambia stato e muove la testina dei nastri di lavoro e di input;
- si legge, oltre al simbolo corrente del nastro di input, anche il simbolo del nastro di lavoro; a questo punto la program machine si comporta secondo uno dei casi appena descritti.

Per semplificare la comprensione dell'output della Program Machine, verrà segnalato nel nastro di output un evento generico tramite il simbolo e_{evento}: ad esempio la lettura di un file verrà segnalata tramite il simbolo e_{read}. Per convenzione ogni **PM** avrà in corrispondenza della prima cella di output $\chi[1]$ l'evento e_{PM}, dove χ è la stringa che denota l'output ed e_{PM} è il codice sorgente della macchina. Ogni lettura dal nastro di input genera un evento di output: se viene letto $\gamma \in \Gamma$, allora verrà generato sul nastro di output l'evento e_γ. Le modifiche del nastro di lavoro vengono segnalate tramite la scrittura sul nastro di output dell'evento $e_{internal}$.
Dalla definizione formale delle **PM** è possibile descrivere in termini matematici la nozione di politica di non interferenza.

Definizione 9.3.2.

Una politica di non interferenza è una funzione $\mathcal{P} : \mathcal{PM} \rightarrow Bool$.

Data una politica di non interferenza P si potrà allora modellare un **detector** per tale politica come segue:

$$\hat{P} : \Gamma^{\omega} \rightarrow Bool$$

dove Γ^{ω} rappresenta l'insieme delle stringhe di output (eventualmente infinite). Affinché $\hat{P}$ sia un detector per la politica di sicurezza, deve essere inoltre soddisfatta la seguente condizione:

$$P(PM) \Leftrightarrow \forall \chi \in \Gamma^{\omega}.\hat{P}(\chi)$$

Ad esempio, si consideri la politica di sicurezza le cui specifiche impediscono la scrittura di una locazione nella regione iniziale della memoria. Ciò è abbastanza ragionevole, se si considera che tale regione di memoria è utilizzata dal solo sistema operativo. Possiamo allora definire formalmente tale politica come segue; sia σ una possibile stringa di input nell'insieme Γ^*:

$$P_{boot}(PM) = (\forall \sigma \in \Gamma^* \; e_{write_i} \notin out(PM(\sigma)), \; i \in [0, \cdots, n])$$

dove la funzione *out* denota l'output dell'esecuzione sulla **PM** e le locazioni che si vogliono proteggere hanno un indirizzo compreso tra 0 e n. La corrispondente funzione detector $\hat{P}$ per P è così definita:

$$\hat{P}_{boot}(\chi) = (e_{write_i} \notin \chi, i \in [0, \cdots, n])$$

9.4 Tipologie di politiche di sicurezza

Daremo adesso una classificazione delle politiche di sicurezza utilizzando le proprietà dei linguaggi (decidibili, ricorsivamente enumerabili, etc.) definite da tali politiche.

9.4.1 Analisi Statica

I meccanismi di *enforcement* che si occupano di rilevare programmi reputati non sicuri prima della loro esecuzione vengono definiti di **analisi statica**. Tali meccanismi devono verificare se il programma preso in analisi soddisfi o meno la politca di sicurezza in un tempo finito. I programmi accettati da tali politiche possono essere mandati in esecuzione, mentre quelli rigettati non dovranno essere eseguiti. Gli esempi che abbiamo preso in esame sono i sistemi di tipi come, per esempio, quello di TAL.
Formalmente, una politica di sicurezza $\mathcal{P}$ è detta *statically enforceable* nel nostro modello se esiste una macchina di Turing $M_{\mathcal{P}}$ che presa in input una program machine M arbitraria tale che $\mathcal{P}(M)$ vale sse $M_{\mathcal{P}}(M)$ accetta (in tempo finito).

Definizione 9.4.1.

Definiamo la classe ***SA*** *come la classe delle politiche per cui esiste un detector che termina in tempo finito su qualsiasi input.*

9.4.2 Execution Monitoring

I *Reference Monitor* e altri meccanismi di sicurezza che operano durante l'esecuzione di un programma vengono detti ***execution monitor*** (EM). Un EM analizza gli eventi di output esibiti da un programma in esecuzione e verifica che questi non violino la politica di sicurezza. L'intervento da parte di un EM potrebbe risultare nella terminazione del programma analizzato oppure in delle operazioni di correzione. Gli esempi più comuni di *Execution Monitoring* includono le liste di controllo degli accessi e il supporto hardware per la protezione della memoria.
Formalmente definiamo una politica di sicurezza definibile tramite EM come segue:

Definizione 9.4.2.

Una politica $\mathcal{P}$ è detta ***EM enforceable*** *se esiste un detector $\hat{\mathcal{P}}$ tale che*

$$
\begin{array}{ll}
M1 & \mathcal{P}(M) \Leftrightarrow \forall \chi \in out(M).\hat{\mathcal{P}}(\chi) \\
M2 & \hat{\mathcal{P}}(\chi[\dots i]) \Rightarrow \forall j \leq i \hat{\mathcal{P}}(\chi[\dots j]) \\
M3 & \neg\hat{\mathcal{P}}(\chi) \Rightarrow \exists i : \neg\hat{\mathcal{P}}(\chi[\dots i]) \\
M4 & \hat{\mathcal{P}}(\chi) \text{ è decidibile se } \chi \text{ è finita.}
\end{array}
$$

La classe delle politiche EM-enforceable prende il nome di ***Mon****.*

9.4.3 Program Rewriting

La riscrittura dei programmi si riferisce a quei meccanismi di enforcement che, in un tempo finito, modificano un programma malizioso prima della sua esecuzione. E' implicita nella riscrittura di programmi la nozione di equivalenza di programmi: la riscrittura non deve alterare la semantica del programma. Dunque le esecuzioni di programmi che hanno subito delle variazioni a causa della loro riscrittura devono avere una qualche corrispondenza con le esecuzioni dell'originale. Formalizziamo questo concetto in termini di una relazione di equivalenza.

Definizione 9.4.3.

Due program machine M_1 e M_2 sono equivalenti ($M_1 \approx M_2$) se, e solo se,

$$\forall \sigma : \sigma \in \Gamma^{\omega} : out(M_1(\sigma)) \approx_{\chi} out(M_2(\sigma))$$

dove i requisiti per $\approx_{\chi}$ sono i seguenti:

$EQ1$	*$\chi_1 \approx_{\chi} \chi_2$ è decidibile qualora χ_1 e χ_2 sono entrambe finite*
$EQ2$	$\chi_1 \approx_{\chi} \chi_2 \Rightarrow \forall i \exists j : \chi_1[\ldots i] \approx_{\chi} \chi_2[\ldots j]$
$EQ3$	*se $\hat{\mathcal{P}}$ è un detector per $\mathcal{P}$ e $\chi_1 \approx_{\chi} \chi_2$ allora $\hat{\mathcal{P}}(\chi_1) \Leftrightarrow \hat{\mathcal{P}}(\chi_2)$*

In particolare un detector che soddisfa la condizione *EQ3* viene detto **consistente** rispetto a $\approx$. Difatti la definizione di tale proprietà impone che un detector non classifica una esecuzione come accettabile e un'altra come non accettabile, quando queste sono equivalenti secondo $\approx_{\chi}$.
Tramite la relazione di equivalenza appena introdotta possiamo dare una definizione formale della classe delle politiche di sicurezza basate sul program rewriting:

Definizione 9.4.4.

Una politica $\mathcal{P}$ è una politica basata sulla riscrittura di programmi se esiste una funzione calcolabile $R : \mathcal{PM} \rightarrow \mathcal{PM}$ tale che per ogni PM M:

(RW1) $\mathcal{P}(R(M))$ *è vera.*

(RW2) $\mathcal{P}(M) \rightarrow M \approx R(M)$.

9.5 Classi di computabilità dei meccanismi di enforcement

In questa sezione prenderemo in considerazione le classi computazionali di appartenenza delle tecniche viste nei capitoli precedenti: analisi statica, politiche monitorabili e riscrittura di codice. Cominciamo ad analizzare le tecniche di analisi statica (AS).

Teorema 9.3. *Le tecniche di analisi statica corrispondono all'insieme delle politiche decidibili.*

Dimostrazione. ($\Rightarrow$) Sia $\mathcal{P}$ una politica di sicurezza appartenente a AS, per definizione deve esistere una Macchina di Turing terminante $M_{\mathcal{P}}$ che prende un encoding di una program machine PM e restituisce in output 0 se $\mathcal{P}$ non vale, 1 altrimenti. Questo però vuol dire che esiste un algoritmo che accetta o rifiuta PM in tempo finito; quindi, $\mathcal{P}$ è una politica decibile.
($\Leftarrow$) Sia $\mathcal{P}$ una politica decidibile allora esiste un algoritmo $\mathcal{A}$ che termina in un tempo finito che implementa $\mathcal{P}$. Per definizione esiste una macchina di Turing TM che codifica $\mathcal{A}$. □

Ora notiamo che i meccanismi di Execution Monitoring sono sostanzialmente pezzi di codice che terminano programmi che potrebbero violare una certa politica. In termini di macchine di Turing, questo si traduce in una TM che prende in input una program machine PM e si ferma se e solo se $P(PM)$ non vale. Notiamo che per definizione $P \in coRE$[1]. Possiamo così dare la seguente definizione.

Definizione 9.5.1.

Sia $\mathcal{P}$ una politica di monitoring allora $\mathcal{P} \in coRE$ sse esiste una macchina di Turing M t.c. $\forall PM$ $M(PM)$ termina sse $\mathcal{P}(PM)$ non vale.

Studiamo la classe computazionale delle politiche di monitoring dimostrando che coincidono proprio con la classe coRE.

Teorema 9.4. *La classe indotta dalle proprietà $EM1..EM4$ coincide con coRE.*

Dimostrazione. ($\Rightarrow$) Mostriamo prima che se una politica di sicurezza soddisfa $EM1..EM4$ allora appartiene a coRE. Sia $\mathcal{P}$ la suddetta politica. Per la proprietà $EM1$ deve esistere un detector $\hat{\mathcal{P}}$ t.c. $\mathcal{P}(PM)$ vale sse $\forall \mathcal{X} \in out(PM)$ $\hat{\mathcal{P}}(\mathcal{X})$ vale. Notiamo che per la proprietà $EM4$, $\hat{\mathcal{P}}(\mathcal{X})$ è decidibile se $\mathcal{X}$ è di lunghezza finita; quindi deve esistere una macchina di Turing $\hat{M}$ che calcola $\hat{\mathcal{P}}$. Costruiamo ora una macchina di Turing $\mathcal{M}$ che termina sse $\mathcal{P}(PM)$ non vale.
$\mathcal{M}(PM) =$

1. genera tutti i prefissi finiti di tracce di output;
2. su ognuno di questi calcola $\hat{\mathcal{P}}$ (o equivalentemente invoca $\hat{M}$);
3. $\mathcal{M}$ si ferma appena ottiene un no da $\hat{M}$.

Dimostriamo che $\mathcal{M}$ termina sse riceve un prefisso che non soddisfa la politica $\mathcal{P}$.

[1] Ricordiamo che coRE è l'inverso della classe di funzioni non ricorsivamente enumerabili e una funzione f è ricorsivamente enumerabile sse $\exists TM$ che riconosce f.

Supponiamo che $\mathcal{P}(PM)$ vale, vuol dire cioè che (per la proprietà $EM1$) $\forall \mathcal{X} \in out(PM)$ $\hat{\mathcal{P}}(\mathcal{X})$ vale. In modo equivalente abbiamo che $\nexists \mathcal{X} \in out(PM)$ $\neg\hat{\mathcal{P}}(\mathcal{X})$ ma tutto ciò significa che $\hat{\mathcal{M}}$ restituisce sempre si e per definizione di $\mathcal{M}$ non si ferma.

Supponiamo ora che $\neg\mathcal{P}(PM)$ dimostriamo che $\mathcal{M}$ termina. Se $\neg\mathcal{P}(PM)$ allora per $EM1$ $\exists \mathcal{X} \in out(PM)$ t.c. $\neg\hat{\mathcal{P}}(PM)$. Ma per $EM3$, $\exists i$ t.c. $\neg\hat{\mathcal{P}}(\mathcal{X}[..i])$; quindi per definizione di $\hat{\mathcal{M}}$ sicuramente $\exists\mathcal{X}\exists i$ t.c. $\hat{\mathcal{M}}$ rigetta $\mathcal{X}[..i]$ così $\mathcal{M}$ termina.

($\Leftarrow$) Dimostriamo ora che se una politica $\mathcal{P} \in coRE$ allora soddisfa $EM1..EM4$. Sia $\mathcal{P}$ la politica in questione, esisterà un detector $\hat{\mathcal{P}}$ per $\mathcal{P}$ che soddisfa $EM2..EM4$. Definiamo $\hat{\mathcal{P}}(\mathcal{X}) \stackrel{\triangle}{=} \mathcal{M}(PM)$ *non termina in al più* $|\mathcal{X}|$ *passi* dove PM è il programma che ha generato $\mathcal{X}$ cioè $\mathcal{X}[1]$. Dimostriamo che $\hat{\mathcal{P}}$ è effettivamente un detector per $\mathcal{P}$ e che soddisfa $EM2..EM4$.

($EM1$)**:** Per definizione $\mathcal{P}$ vale sse $\mathcal{M}(PM)$ non termina quindi $\forall \mathcal{X}$ $\mathcal{M}(PM)$ non termina in $|\mathcal{X}|$ passi. Per definizione di $\hat{\mathcal{P}}$ abbiamo che $\forall \mathcal{X}$ $\hat{\mathcal{P}}(\mathcal{X})$ vale, così possiamo concludere dicendo che $\hat{\mathcal{P}}$ è un detector per $\mathcal{P}$.

($EM2$)**:** Se vale $\hat{\mathcal{P}}(\mathcal{X})$ allora $\mathcal{M}(PM)$ non termina in m passi, quindi per ogni $i < m$ non termina in al più i passi.

($EM3$)**:** Per definizione abbiamo che se $\neg\hat{\mathcal{P}}(\mathcal{X})$ $\mathcal{M}(PM)$ termina in al più $|\mathcal{X}|$ passi ma questo non è altro che dire che $\exists i$ t.c. $\neg\hat{\mathcal{P}}(\mathcal{X})[..i]$

($EM4$)**:** Dobbiamo dimostrare che se $\mathcal{X}$ è finita allora $\hat{\mathcal{P}}(\mathcal{X})$ è decibile. Ma se $\mathcal{X}$ è finita allora $\hat{\mathcal{P}}(\mathcal{X})$ può essere simulata con $\mathcal{M}(PM)$ in al più $|\mathcal{X}|$ passi.

□

9.6 Relazioni tra le classi di computabilità

Per concludere lo studio effettuato sulle politiche di sicurezza e sulle classi di computazione da esse indotte, si vogliono studiare le relazioni insiemistiche che intercorrono tra di esse.
In particolare, si vuole dimostrare che tutte le politiche di analisi statica sono implementabili tramite execution monitor e program rewriting, vale a dire la classe **SA** è inclusa sia nella classe **Mon** che nella classe **RW**. Inoltre verrà notato che non intercorre alcuna relazione di inclusione, nè tanto meno di uguaglianza, tra le classi **Mon** e **RW**. Esistono pertanto delle politiche di sicurezza implementabili tramite Execution Monitor ma non tramite Program Rewriting, e viceversa.
Volendo dare una rappresentazione grafica delle relazioni che intercorrono tra queste tre classi computazionali, si ottiene il grafico della figura 9.6. Per essere precisi, l'inclusione di SA in RW non è completa: esiste una politica (quella che rifiuta sempre) che appartiene a Mon e SA, ma non a RW.

Iniziamo a dimostrare formalmente la gerarchia di figura 9.6.

Corollario 9.5. $SA \subset Mon$.

Dimostrazione. Da risultati classici di teoria della calcolabilità (per cui i problemi decidibili sono strettamentee inclusi nei co-RE), usando i teoremi 9.3 e 9.4. □

Teorema 9.6. $SA \setminus \{Unsat\} \subseteq RW$.

Dimostrazione. Sia $P \in SA \setminus \{Unsat\}$; poichè $P \neq Unsat$, esiste una program machine M per cui vale $P(M)$. Definiamo una funzione totale R su program machines tale che

$$R(M') = \begin{cases} M' & \text{se vale } P(M') \\ M & \text{altrimenti} \end{cases}$$

Chiaramente R è decidibile, poichè lo è P (usando il teroema 9.3). Inoltre, R soddisfa (RW1) per costruzione e (RW2) per riflessività di $\approx$. □

Teorema 9.7. $(RW \cap Mon) \setminus SA \neq \emptyset$.

Dimostrazione. Sia e un certo evento e si consideri la politica P_e che rifiuta tutte e sole le macchine che esibiscono l'evento e. Tale politica non appartiene a SA: essa è strettamente collegata al problema della fermata, che è noto essere indecidibile. Al contrario, è banale mostrare che $P_e \in Mon$. E' anche facile mostrare che $P \in RW$: basta definire la funzione R in modo che trascrive ogni macchina M in una macchina che, ad ogni passo, simula il comportamento di M, vede se l'evento che ciò produrrebbe è e ed, in tal caso, termina immediatamente la sua computazione. □

Teorema 9.8. $RW \setminus Mon \neq \emptyset$.

Dimostrazione. Sia $\approx$ una relazione che non appartiene a co-RE; allora, esiste una program machine M' per cui decidere $M \approx M'$ per una qualsiasi M risulta

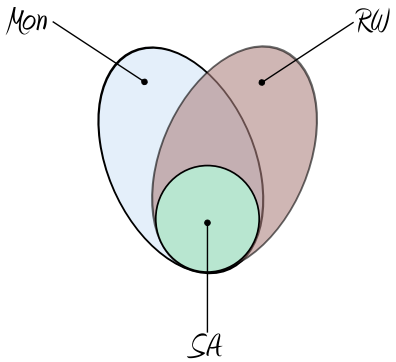

Figura 9.2: Gerarchia delle classi di complessità

non essere co-RE. Definiamo allora $P(M)$ come $M \approx M'$. Chiaramente, $P \notin$ co-RE ($= Mon$), mentre $P \in RW$: infatti, basta prendere come funzione R la funzione che restituisce sempre M' (è banalmente decidibile e, per costruzione, soddisfa (RW1) e (RW2)). □

Per completare la gerarchia mostrata, ci resta da dimostrare che esistono politiche di Mon che però non appartengono a RW.

Teorema 9.9. *$Mon \setminus RW \neq \emptyset$.*

Questo risultato è tutt'altro che banale e utilizza delle nozioni nuove che ora andiamo a definire. Introduciamo dapprima una distinzione tra le tipologie di detector alla base di una determinata politica di sicurezza.

Definizione 9.6.1.

*Un detector $\hat{\mathcal{P}}$ è detto **benevolo** qualora esista una procedura di decisione $A_{\hat{\mathcal{P}}}$ su esecuzioni finite tale che per ogni Program Machine M:*

$B1$ $(\exists \chi \in out(M)$ *t.c.* $\neg\hat{\mathcal{P}}(\chi)) \Rightarrow (\forall \chi \in out_{fin}(M)\ \neg\hat{\mathcal{P}}(\chi) \Rightarrow A_{\hat{\mathcal{P}}}(\chi)$ *rigetta*)

$B2$ $\forall \chi \in out(M)\hat{\mathcal{P}}(\chi) \Rightarrow \forall \chi \in out_{fin}(M) A_{\hat{\mathcal{P}}}(\chi)$ *accetta*

La prima condizione stabilisce che, se è possibile trovare un output (anche infinito) tale che la proprietà del detector su tale stringa non è verificata per la program machine M, allora per ogni output **finito** per il quale non valga la proprietà $\hat{\mathcal{P}}$ la macchina di decisione $A_{\hat{\mathcal{P}}}$ rigetta. La seconda condizione invece asserisce che, qualora la proprietà $\hat{\mathcal{P}}$ risulta valida per ogni output (finito e infinito) allora per ogni sequenza di output finita la macchina di decisione $A_{\hat{\mathcal{P}}}$ accetta.
L'introduzione dei detector benevoli ci permette di ottenere delle interessanti informazioni sulle classi computazionali definite per le politiche di sicurezza. Una prima proprietà, di ovvia utilità, permette di stabilire una relazione che intercorre tra un detector che soddisfa le proprietà *M 1-4* e i detector benevoli:

Lemma 9.10. *Sia $\mathcal{P}$ una politica che ammette un detector $\hat{\mathcal{P}}$ che soddisfa le proprietà $M1-4$. Sia inoltre $\mathcal{P} \in RW$, quindi consistente rispetto a $\approx$. Allora $\hat{\mathcal{P}}$ è benevolo.*

Dimostrazione. Per ipotesi la politica $\mathcal{P}$ soddisfa le condizioni $M1-4$. Per definizione allora $\mathcal{P} \in Mon$, e per quanto dimostrato nel teorema (9.4), si ha che $\mathcal{P} \in coRE$. Allora esiste una macchina di Turing TM tale che, per ogni program machine M, si ha che $Tm(M)$ termina se e solo se $\neg\mathcal{P}(M)$. Ciò risulta ovvio, in quanto tale macchina di Turing si ferma non appena riscontra che la politica viene violata.
Inoltre, poiché $\mathcal{P} \in RW$, esiste una funzione R (decidibile) tale che:

- $\mathcal{P}(R(M))$
- $\mathcal{P}(M) \Rightarrow M \approx R(M)$

Ricordiamo adesso che il primo evento di output generato da una Program Machine è un evento che identifica in maniera univoca il codice sorgente della program machine stessa. Dunque, presa una qualsiasi sequenza di eventi di output χ, allora è ovvio che $\chi[1] = M$. Inoltre tale output χ dovrà essere stato generato a partire da una stringa di input σ, dunque $\exists\sigma : M(\sigma) = \chi$.
Sulla base di tali informazioni viene definita la macchina decisionale per il detector $\hat{\mathcal{P}}$:

$\forall i \geq 1$:

- Se $\chi \approx_\chi R(M(\sigma))[\dots i] \Rightarrow$ accetta.
- Se $Tm(M)$ si ferma dopo i passi, allora rigetta.

Si vuole mostrare che tale macchina di decisione soddisfa le proprietà necessarie affinché il detector risulti benevolo. Mostriamo pertanto che le proprietà $B1-2$ vengono soddisfatte e che tale macchina termina sempre:

B2) Per ipotesi $\hat{\mathcal{P}}$ soddisfa la proprietà $M1$ ed è consistente rispetto a $\approx$. Allora ne risulta che $\mathcal{P}(M)$ e, poiché $\mathcal{P} \in RW$, risulta che $M \approx R(M)$. Per definizione di $\approx$, si ha che $M(\sigma) \approx_\chi R(M)(\sigma)$ e, poiché $M(\sigma) = \chi$, risulta $\chi \approx_\chi R(M)(\sigma)$. Poichè χ è prefisso di sé stessa, allora $\exists i : \chi \approx_\chi R(M)(\sigma)[\dots i]$. Per come è stata definita la macchina decisionale $A_{\hat{\mathcal{P}}}$, il primo if è verificato, mentre il secondo non lo è, a causa della definizione di $Tm(M)$, perchè la politica è rispettata. Da ciò risulta che $A_{\hat{\mathcal{P}}}(\chi)$ accetta.

T) Mostriamo adesso che l'algoritmo termina. Distinguiamo due casi, a seconda che valga o meno $\mathcal{P}(M)$. Nel caso in cui tale ipotesi risulti soddisfatta, allora per la proprietà $B2$ l'algoritmo banalmente termina. Nel caso contrario ($\neg\mathcal{P}(M)$), per definizione di $coRE$, $TM(M)$ termina, e allora esiste un indice i per il quale la macchina si ferma. Dunque, dalla descrizione che si è data della macchina $A_{\hat{\mathcal{P}}}$, si avrà che tale macchina termina. Tale terminazione deriva dalla seconda condizione che è stata stabilita al momento di definire la macchina $A_{\hat{\mathcal{P}}}$, tuttavia si noti che ad un passo precedente potrebbe essere risultata vera la prima condizione; tuttavia in tal caso la proprietà $B2$ precedentemente dimostrata assicura nuovamente la terminazione della macchina.

B1) Per completare la dimostrazione è infine necessario dimostrare che, qualora $A_{\hat{\mathcal{P}}}$ accetta, allora risulta vera la proprietà $\mathcal{P}(M)$. Affinché $A_{\hat{\mathcal{P}}}$ accetti, deve esistere un indice i tale che $\chi \approx_\chi R(M)(\sigma)[\dots i]$; Allora, poiché $\mathcal{P} \in RW$, banalmente risulta valida la proprietà $\mathcal{P}(R(M))$. Ciò equivale a dire che $\forall\chi' \in out(R(M))$ viene soddisfatta la proprietà $\hat{\mathcal{P}}(\chi')$. In particolare, preso $\chi' = R(M)(\sigma)$, si ha che $\hat{\mathcal{P}}(R(M)(\sigma))$ risulta vera. Dunque per la proprietà $M2$ si ha che $\forall j\ \hat{\mathcal{P}}(R(PM)(\sigma[\dots j]))$. Da qui

risulta $\forall \chi \in out(M)\ \hat{\mathcal{P}}(\chi)$. Tale asserto coincide, per la proprietà $M1$, con $\mathcal{P}(M)$.

Supponiamo che sia $\neg\mathcal{P}(M)$. Allora $\exists\chi.\neg\hat{\mathcal{P}}(\chi)$, e la tesi risulta immediata da quanto discusso precedentemente.

□

Corollario 9.11. *Sia $\mathcal{P}$ una politica che ammette un detector $\hat{\mathcal{P}}$ che soddisfa le proprietà $M1-4$, consistente rispetto a $\approx$ e* ***non benevolo****. Allora $\mathcal{P} \notin RW$.*

Se dimostriamo che esiste una politica come quella descritta nelle ipotesi del corollario 9.11, allora il teorema 9.9 risulta essere dimostrato.

Dimostrazione del Teorema 9.9. Definiamo il seguente detector:

$$\hat{\mathcal{P}}(\chi) \text{ risulta vera sse } \chi[1] \text{ non termina su nessun input.}$$

Sia dunque $\mathcal{P}$ la politica indotta da $\hat{\mathcal{P}}$; le proprietà $M1-4$ risultano banalmente soddisfatte. Dimostriamo adesso che il detector non è benevolo. Per assurdo, se $\hat{\mathcal{P}}$ fosse benevolo allora $\exists A_{\hat{\mathcal{P}}}$ tale da soddisfare le condizioni $B1-2$; dimostrano che, se esistesse, l'algoritmo risolverebbe il problema della fermata. Sia definisca infatti la seguente macchina di Turing:

- riceve la program machine M;
- calcola $A_{\hat{\mathcal{P}}}(M(\sigma))[1]$ con $\sigma \in \Gamma^*$ (viene presa in input una qualsiasi stringa finita);
- se accetta allora non termina altrimenti termina.

Se la program machine M non termina allora $\hat{\mathcal{P}}$ è verificata e la macchina di Turing appena definita accetta poichè vale la proprietà $B2$. In caso contrario tale macchina di Turing rifiuta in quanto vale la proprietà $B1$. Poichè è nota l'indecidibilità del problema della fermata allora ne risulta che $\hat{\mathcal{P}}$ non può essere benevolo. □

Bibliografia

[1] Fred B. Schneider, J. Gregory Morrisett, and Robert Harper. A language-based approach to security. In *Informatics - 10 Years Back. 10 Years Ahead.*, volume 2000 of *LNCS*, pages 86–101. Springer, 2001.

[2] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communication*, 21:5–19, 2003.

[3] Dennis M. Volpano and Geoffrey Smith. Language issues in mobile program security. In *Mobile Agents and Security*, volume 1419 of *LNCS*, pages 25–43. Springer, 1998.

[4] Geoffrey Smith Dennis Volpano and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2/3):167–188, 1996.

[5] Geoffrey Smith Dennis Volpano. Secure information-flow in a multi-threaded imperative language. In *Proc. of POPL*, pages 355–364. ACM, 1998.

[6] Geoffrey Smith Dennis Volpano. Verifying secrets and relative secrecy. In *Proc. of POPL*, pages 268–276. ACM, 2000.

[7] Sebastian Hunt David Clark and Pasquale Malacaria. Quantitative information flow, relations and polymorphic types. *Journal of Logic and Computation*, 15(2):181–199, 2005.

[8] David Walker. *An Introduction to Typed Assembly Language.* http://www.cs.cmu.edu/~dpw/papers/tal-intro.ps.

[9] Fred B. Schneider. Enforceable security policies. *ACM Transaction on Information System Security*, 3(1):30–50, 2000.

[10] Ulfar Erlingsson Fred Schneider. Sasi enforcement of security policies: A retrospective. In *WNSP: New Security Paradigms Workshop*, 1999.

[11] Fred B. Schneider Kevin W. Hamlen, Greg Morrisett. Computability classes for enforcement mechanisms. *ACM Transactions on Programming Languages and Systems*, 28(1):175–205, 2006.

[12] Dennis M. Volpano and Geoffrey Smith. Probabilistic noninterference in a concurrent language. *Journal of Computer Security*, 7(1), 1999.