SAPIENZA
UNIVERSITÀ DI ROMA

Facoltà di Scienze MM.FF.NN.
Corso di laurea in Informatica

Tesi di Laurea Specialistica

Un Approccio Incrementale alla Semantica della Sicurezza

Relatore:
Pietro Cenciarelli

Candidato:
Andrea Cerone (697074)

A.A 2007/2008

Un manipolatore formale in
matematica spesso prova la
sconfortante sensazione che la sua
matita lo superi in intelligenza.

Howard W. Eves, 1911-2004

Indice

0 Introduzione

L'obiettivo di questa tesi è lo sviluppo di una **Semantica Denotazionale** modulare oer un semplice linguaggio di programmazione imperativo, che caturi aspetti di **Sicurezza**.
Per raggiungere tale obiettivo, si è scelto di utilizzare la **teoria delle categorie** e gli strumenti che vengono con essa. L'uso di tale teoria permette di operare ad un livello di astrazione tale da poter analizzare i programmi di un dato linguaggio dal punto di vista della loro interpretazione in un modello matematico, tralasciando tutti gli aspetti relativi alla sintassi.
Nella teoria delle categorie sono presenti inoltre gli strumenti necessari per analizzare gli effetti computazionali associati ai programmi. Gli effetti computazionali, quali ad esempio le eccezioni e l'I/O, possono essere infatti rappresentati come **monadi** presenti nella categoria che si è scelta per interpretare il linguaggio.
Con l'introduzione dei **costruttori semantici** è possibile inoltre affrontare la questione della semantica in maniera modulare: dato un linguaggio nel quale è presente un determinato effetto computazionale, è possibile definire per esso una semantica denotazionale a partire da una semantica preesistente sulla quale non era definito il comportamento dei programmi rispettivamente all'effetto computazionale considerato.
Monadi e costruttori semantici verranno ampiamente utilizzati in questa tesi per integrare aspetti di sicurezza nella semantica di un semplice linguaggio di programmazione imperativo.

0.1 Aspetti Semantici

La **Semantica dei Linguaggi di Programmazione** occupa una posizione centrale nell'area di studio dei **Metodi Formali dell'Informatica**: in particolare, la **semantica denotazionale** viene utilizzata per definire il significato dei programmi interpretandoli in oggetti matematici.
La sintassi di un linguaggio di programmazione viene, definita opportunamente da una grammatica ed, eventualmente, da un sistema di tipi, stabilisce l'insieme dei programmi sintatticamente corretti per tale linguaggio. Tale grammatica non dice nulla su quale sia

il risultato dell'esecuzione di un programma all'interno di un calcolatore, o di quale sia la funzione matematica che descrive il comportamento di un programma. La semantica, invece, permette di interpretare i programmi all'interno di un modello matematico: tale modello dovrà presentare la struttura necessaria che permetta di descrivere il comportamento dei programmi.

La **Teoria delle Categorie** viene utilizzata in informatica per definire la semantica denotazionale di un linguaggio di programmazione: una categoria è data da una collezione di oggetti, corrispondenti a strutture matematiche, e da una collezione di morfismi, corrispondenti a trasformazioni di oggetti, ciascuno avente un dominio e un codominio scelti tra gli oggetti. Un morfismo f avente per dominio un oggetto A e per codominio un oggetto B viene indicato con la notazione $f : A \to B$. Inoltre, viene definita un'operazione di composizione per i morfismi di una categoria, per la quale si richiede che valga la proprietà associativa. Le proprietà che gli oggetti e i morfismi di una categoria soddisfano danno la struttura al mondo matematico considerato: in letteratura sono state analizzate molte costruzioni e proprietà che possono presentarsi all'interno di una categoria, e la loro rilevanza nella semantica denotazionale. Dati un linguaggio di programmazione per il quale sia stato definito un sistema di tipi ed una categoria che presenti la struttura necessaria per interpretare i programmi, la semantica denotazionale di tale linguaggio viene definita fornendo una funzione di interpretazione $[\![\cdot]\!]$ che associ ad ogni tipo un oggetto della categoria, e ad ogni regola di inferenza del sistema di tipi che definisce il linguaggio un morfismo all'interno della categoria stessa.
Uno dei più importanti contributi alla semantica è stato dato da **Dana Scott** con l'introduzione della **Teoria dei Domini**, che studia matematiche idonee all'interpretazione di linguaggi di programmazione che permettono la ricorsione (condizione necessaria affinché si possa ottenere un **Linguaggio Turing Completo**.
Si consideri un insieme C sul quale viene definito un ordinamento parziale $\sqsubseteq$: tale insieme viene detto completo (**Complete Partially Ordered set**, o semplicemente **CPO**) se, comunque si scelga una catena $c_0 \sqsubseteq c_1 \sqsubseteq \cdots$ tra gli elementi di C, esiste $c = \bigsqcup_n c_n \in C$ tale che

$$\begin{gathered} \forall n.c_n \sqsubseteq c \\ \forall c'.\forall n.d_n \sqsubseteq c' \quad \Rightarrow \quad c' \sqsubseteq c \end{gathered}$$

L'elemento $\bigsqcup_n c_n$ prende il nome di **least upper bound** della catena.
Un CPO nel quale ogni elemento può essere ottenuto come least upper bound di una catena di **elementi compatti**[1] viene detto **Dominio**. La teoria dei domini di Scott viene utilizzata per la risoluzione delle equazioni ricorsive che possono presentarsi nell'interpretazione di un programma. Si consideri ad esempio un programma del tipo while E do C: esso può essere riscritto come

if E then (C; while E do C) else skip

[1] un elemento di un insieme parzialmente ordinato è detto compatto se non può essere ottenuto come minimo confine superiore di una catena nel quale non sia presente l'elemento stesso

Si ottiene l'equazione ricorsiva

$$\text{while E do C} = \text{if E then (C; while E do C) else skip}$$

La soluzione di tale equazione è data da

$$\text{while E do C } = \mu f.\text{if E then C };f \text{ else skip}$$

Dove, la notazione $\mu x.f(x)$, con $f : A \to A$, denota il minimo elemento x isomorfo ad $f(x)$.
Se l'interpretazione di tale programma avviene all'interno di un dominio, è possibile approssimare la soluzione di tale equazione definendo una catena $d_0 \sqsubseteq d_1 \sqsubseteq \cdots$ di possibili valori corrispondenti alla soluzione dell'equazione, in cui d_0 corrisponde all'elemento minimo del dominio, e d_n viene ottenuto sostituendo il termine f con d_{n-1} nell'equazione

$$\text{if E then (C; } f\text{) else skip}$$

La soluzione dell'equazione corrisponderà difatti con $\bigsqcup_n d_n$, appartenente al dominio per definizione.
L'uso della teoria dei domini ha permesso di definire la semantica dei linguaggi di programmazione per i quali fosse definita la ricorsione all'interno della categoria **CPO**, in cui gli oggetti sono appunto i CPO, e i morfismi sono le funzioni continue, ovvero funzioni tra CPO che preservano l'ordinamento e i least upper bounds delle catene.

Un secondo strumento importante della semantica, che verrà utilizzato ampiamente nella stesura della tesi, è dato dal **λ-calcolo computazionale**, introdotto da **Eugenio Moggi** in [Mog89b].
In tale lavoro **Moggi** introduce formalmente il concetto di computazione su di un valore utilizzando le **Triple di Kleisli**: una tripla di Kleisli è data da un'associazione tra oggetti di una categoria T, equipaggiata con una famiglia di morfismi $\eta_A : A \to TA$ e un'operatore $\_^*$ tale che, se $f : A \to TB$, allora $f^* : TA \to TB$. Una computazione su un valore di tipo A viene definita come un elemento dell'oggetto TA.
Il λ-calcolo computazionale è un metalinguaggio all'interno del quale vengono definiti due operatori, *let* e *val*, utilizzati rispettivamente per la composizione di computazioni e la trasformazione di valori in computazioni. L'introduzione di tale strumento ha permesso la definizione della semantica di linguaggi di programmazione per i quali siano definiti differenti effetti computazionali, quali ad esempio le **eccezioni** o la **gestione dell'I/O** di programmi.
Infine, in [Mog91] Moggi propone l'uso dei costruttori semantici per definire nuovi effetti computazionali. Un costruttore semantico è una funzione che, presa in input una tripla di Kleisli, ne restituisce una nuova in cui sia stato introdotto un nuovo effetto computazionale: tali strumenti hanno permesso di rendere la semantica dei linguaggi di programmazione modulare, consentendo così lo studio di linguaggi di programmazione complessi, ricchi di effetti computazionali. In questa tesi si è affrontato il problema di definire la semantica di un linguaggio di programmazione imperativo, incorporando in essa il concetto di **Sicurezza**. A tale scopo sono state utilizzate le triple di Kleisli e i costruttori semantici

per l'introduzione di uno stato di errore, che può essere restituito come risultato di una computazione: tale stato di errore indicherà una potenziale violazione della politica di sicurezza considerata. Inoltre, la teoria dei domini e l'operatore di punto fisso μ permettono di risolvere le equazioni ricorsive che si incontreranno nella definizione di tale semantica.

0.2 Aspetti di Sicurezza

La sicurezza di un sistema informatico viene definita da un insieme di specifiche, detto **politica di sicurezza**. Tali specifiche possono essere definite a livello sia descrittivo che formale, e stabiliscono quali azioni siano permesse all'interno di un sistema. Nella definizione di una politica di sicurezza, viene fatta distinzione tra **oggetti**, entità passive presenti all'interno del sistema (quali ad esempio files e record di basi di dati), e **soggetti**, che rappresentano invece le entità attive (quali ad esempio utenti e processi).

Nel campo della sicurezza, le politiche vengono catalogate in tre diverse tipologie, in base agli aspetti di sicurezza che vengono affrontati. Tali aspetti sono i seguenti:

Confidenzialità: gli oggetti devono poter essere acceduti solo da quei soggetti che dispongono dei privilegi necessari.

Integrità: gli oggetti devono poter essere alterati solo da quei soggetti che dispongono dei privilegi necessari: inoltre, la fonte originaria dell'oggetto non deve poter essere alterata dai soggetti che non ne hanno diritto.

Disponibilità: gli oggetti devono poter essere sempre acceduti da chi dispone i permessi necessari.

La **Language Based Security** è un'area dell'informatica che si occupa dello studio della sicurezza dei programmi: le politiche di sicurezza vengono definite stabilendo delle proprietà che un programma deve soddisfare.

Uno degli argomenti più studiati nella Language Based Security è dato dal problema dell'**Information Flow**: nella sua definizione più semplice la memoria di un calcolatore viene partizionata in due zone, rispettivamente dette **Memoria Alta** e **Memoria Bassa**, ed una politica di sicurezza stabilisce che le informazioni contenute nella zona di memoria alta non devono essere trasferite, nè completamente nè parzialmente, nella zona di memoria bassa. Questo è un tipico esempio di politica di **Confidenzialità**, in quanto generalmente si assume che la memoria alta contenga delle informazioni **confidenziali**, mentre la memoria bassa contiene informazioni di pubblico dominio.
Una prima generalizzazione del problema si ottiene partizionando la memoria in più aree, e stabilendo un ordinamento parziale sulle zone di memoria in maniera tale da formare un **Lattice**[2]: considerata tale suddivisione, la politica di sicurezza stabilisce che le informazioni possono fluire da una zona di memoria Z_1 ad un'area di memoria Z_2 se, e solo

[2] l'utilità nell'utilizzare i lattice di sicurezza deriva dal poter selezionare il livello di sicurezza necessario

se, nell'ordinamento del lattice si ha $Z_1 \sqsubseteq Z_2$. In questa generalizzazione possono essere trovate forti analogie con il modello **Bell-La Padula** per l'enforcing della confidenzialità in un sistema informatico: in tale modello viene definito un lattice di livelli di sicurezza, e viene associato un livello di sicurezza ad ogni oggetto e ad ogni soggetto. Vengono quindi stabilite le seguenti regole:

Simple Security Condition: Un soggetto S con livello di sicurezza σ_1 può leggere le informazioni contenute in un oggetto O con livello di sicurezza σ_2 se, e solo se, $\sigma_2 \sqsubseteq s_1$.

***-Property:** Un soggetto S con livello di sicurezza σ_1 può scrivere in un oggetto O con livello di sicurezza σ_2 se, e solo se, $s_1 \sqsubseteq s_2$.

In letteratura sono stati proposti molti **Meccanismi di Sicurezza** in grado di assicurare che la politica dell'information flow non venga violata. Nessuno dei meccanismi proposti è però in grado di distinguere con precisione tra programmi **interferenti** (in cui esistono esecuzioni che portano ad un flusso di informazioni dalla memoria alta alla memoria bassa) e programmi non interferenti. Difatti, si incorre spesso in dei **falsi positivi**, ovvero programmi non interferenti che vengono classificati dal meccanismo di sicurezza come potenzialmente insicuri. Nel corso della tesi verrà mostrata l'impossibilità di poter costruire un tale meccanismo di sicurezza.
Esistono ulteriori generalizzazioni del problema dell'information flow, in quanto è possibile ottenere dei flussi di informazione pur non trasferendo alcuna quantità di informazione dalla memoria alta alla memoria bassa: tra i più noti in letteratura vengono evidenziati i **termination channel**, in cui informazione su una variabile residente in una zona di memoria alta può essere ottenuta osservando la non terminazione di un programma, e i **Time Channel**, ottenuti osservando il tempo di esecuzione di un programma in diversi stati di memoria. Come esempio di time channel, si consideri il seguente programma, in cui h è una variabile appartenente a una zona di memoria alta:

```
1  while (h)
2      h := h−1;
3      sleep(1)
```

Il lavoro svolto in questa tesi è uno studio preliminare del problema, che considera il problema dell'information flow nella sua forma più semplice. In futuro, si vorrà estendere il lavoro svolto, definendo nuove semantiche che tengano conto di altri aspetti quali termination channel e timing channel. Inoltre, ci si chiede se sia possibile generalizzare il lavoro svolto al punto di riuscire a definire una semantica in cui lo stato di errore coincida con una violazione di una generica politica di sicurezza.

per accedere a due oggetti con livelli di sicurezza non confrontabili tra loro: in tale caso, infatti, il livello di sicurezza necessario per accedere ad entrambi gli oggetti è dato dal minimo comune maggiorante dei loro livelli di sicurezza

0.3 Lavoro Svolto

Viene di seguito effettuato un resoconto sul lavoro svolto, esplicitando quali argomenti verranno trattati in ogni capitolo:

Richiami di teoria delle categorie: Vengono introdotti la teoria delle categorie e le costruzioni che verranno utilizzate nel corso dello sviluppo della tesi. La stesura di questo capitolo è stata effettuata facendo riferimento a [Pie91] e [Lan98].

Semantica Categoriale: Si analizza l'uso della teoria delle categorie per la definizione della semantica denotazionale di un linguaggio di programmazione. Tramite un esempio tratto da [Pie91] si mostrerà la semantica denotazionale definita per il λ-calcolo tipato semplice. Viene inoltre mostrato il lavoro effettuato da Moggi in [Mog89b], introducendo il λ-calcolo computazionale e la sua interpretazione canonica in una categoria.

Modelli di Computazione: In questo capitolo vengono mostrati alcuni esempi di monadi computazionalmente rilevanti, e vengono introdotti i costruttori semantici, tramite i quali è possibile ottenere un approccio modulare alla semantica denotazionale. In particolare, vengono analizzati i seguenti effetti computazionali: side effects, non terminazione, errori e context readers. La dimostrazione delle proposizioni è stata data in via originale, in quanto non è stato possibile reperirle negli articoli usati come materiale di riferimento.

Semantica di un linguaggio imperativo: Utilizzando gli strumenti introdotti nei primi tre capitoli, viene mostrato il lavoro necessario per definire la semantica denotazionale di un semplice linguaggio di programmazione imperativo. Il lavoro presente in questo capitolo è stato sviluppato individualmente, facendo riferimento a [Cen96] per i metodi di applicazione degli strumenti utilizzati.

Monadi Nella Language Based Security: Vengono introdotte le nozioni principali di sicurezza informatica, reperite in [Bis03], e il problema dell'information flow, illustrato in [SM03]. Dopo aver mostrato i metodi di risoluzione del problema proposti in letteratura, viene esposto un lavoro orginale nel quale viene definita la semantica denotazionale di un semplice linguaggio di programmazione imperativo, includendo al suo interno il concetto di sicurezza tramite la definizione di uno stato di errore. Si mostrerà inoltre che la semantica ottenuta è sound, ovvero che l'interpretazione di un programma eseguito in una configurazione che genera flusso di informazioni corrisponde allo stato di errore introdotto.

Conclusioni: Viene effettuata una analisi dei risultati ottenuti e su quali direzioni si debbano prendere in futuro per ottenere dei risultati di rilevante interesse.

Nello stendere questa tesi, ho assunto che il lettore abbia le conoscenze che vengono tipicamente acquisite nei corsi propedeutici al percorso in **Linguaggi e Metodi Formali** del corso di studi in Informatica Specialistica. Una conoscenza preliminare della teoria delle categorie e della sicurezza basata sui linguaggi può essere di aiuto, ma non è strettamente necessaria.

1

Richiami di Teoria delle Categorie

Per comprendere gli strumenti che verranno utilizzati all'interno di questa tesi è necessario introdurre i fondamenti della **teoria delle categorie**. Tale teoria risulta essere estremamente vasta, pertanto verranno introdotti solamente i concetti e le costruzioni necessari allo sviluppo degli strumenti che verranno illustrati nel seguito di questo lavoro.

Gli strumenti formalizzati all'interno della teoria delle categorie permettono di lavorare, in maniera astratta, su diversi tipi di strutture matematiche e sulle trasformazioni utilizzate per la loro manipolazione. Un esempio classicamente riportato in letteratura è quello della categoria degli insiemi: in tale categoria la struttura matematica considerata è l'insieme, mentre le trasformazioni che permettono di trasformare un insieme in un altro sono le funzioni.
All'interno di una categoria le singole istanze della struttura matematica presa in considerazione prendono il nome di **oggetti**, mentre le trasormazioni utilizzate per manipolare tali strutture vengono dette **morfismi** o **frecce**.
Nell'informatica la teoria delle categorie ha trovato applicazioni soprattutto nel contesto della **semantica**. Fissati un modello computazionale ed una categoria, si costruirà una funzione di interpretazione che associ, ad ogni termine del modello di computazione scelto, un oggetto all'interno della categoria fissata. Strumenti avanzati della teoria permettono inoltre di rendere tale semantica modulare, così da poter estendere un modello di calcolo con degli ulteriori effetti computazionali, definendone una nuova intepretazione a partire dalla semantica data.

1.1 Categorie

Una categoria $\mathcal{C}$ è costituita da una collezione di oggetti $Obj_{\mathcal{C}}$ e da una collezione di morfismi $mor_{\mathcal{C}}$. Ad ogni morfismo f vengono associati un dominio $dom\ f \in Obj_{\mathcal{C}}$ e un codominio $cod\ f \in Ojb_{\mathcal{C}}$: ogni morfismo può essere rappresentato a partire dal suo dominio e dal suo codominio, tramite la notazione

$$f = dom\ f \to cod\ f$$

Ogni categoria deve essere inoltre equipaggiata con un'operazione $\circ : mor_{\mathcal{C}} \times mor_{\mathcal{C}} \hookrightarrow mor_{\mathcal{C}}$ che permetta di comporre due morfismi: tale operazione viene definita solamente per quelle coppie $< g,f >$ di morfismi tali che $cod\ f = dom\ g$. Il risultato sarà dato da un morfismo $g \circ f : dom\ f \rightarrow cod\ g$: la composizione $g \circ f$ di due morfismi viene spesso indicata in informatica tramite la notazione $f;g$. Infine, per ogni oggetto A di una categoria deve essere definito un particolare morfismo, detto identità di A, tale che la composizione di quest'ultimo con un qualsiasi morfismo f dia come risultato il morfismo f stesso.
Formalizzando i concetti presentati, si ottiene la seguente definizione:

Definizione 1.1.1.

Categorie

Una categoria $\mathcal{C}$ è una collezione di oggetti $Obj_{\mathcal{C}}$ e una collezione di morfismi $mor_{\mathcal{C}}$, equipaggiati con un'operazione di composizione $\circ$ tale che:

- $f : A \rightarrow B, g : B \rightarrow C \Rightarrow g \circ f : A \rightarrow C$.
- *Dati tre morfismi $f : A \rightarrow B$, $g : B \rightarrow C$, $h : C \rightarrow D$, l'uguaglianza*

$$h \circ (g \circ f) = (h \circ g) \circ f \tag{1.1.1}$$

  risulta soddisfatta

- *Dato $A \in Obj_A$ esiste un morfismo $id_A : A \rightarrow A$ tale che*

$$\forall f : B \rightarrow A . id_A \circ f = f \tag{1.1.2}$$
$$\forall g : A \rightarrow B . g \circ id_A = g \tag{1.1.3}$$

Data una categoria $\mathcal{C}$ e due suoi oggetti $A,B \in Obj_{\mathcal{C}}$, l'insieme $\{f \in Mor_{\mathcal{C}} \mid domf = A, codf = B\}$ dei morfismi che hanno come dominio A e come codominio B viene tipicamente denotato con $\mathcal{C}[A,B]$[1].

1.1.1 Esempi di categorie

Per chiarire il concetto di categoria, verranno presentati alcuni esempi di strutture matematiche che possono essere rappresentate all'interno della teoria.
Un primo esempio è dato dalla categoria degli insiemi **Set**, i cui oggetti sono gli insiemi e i cui morfismi sono le funzioni totali. Chiaramente, l'operazione $\circ$ corrisponde alla ben nota composizione di funzioni, mentre dato un insieme A, la funzione id:

[1] Un insieme di morfismi, quale ad esempio $\mathcal{C}[A,B]$, viene spesso indicato in letteratura con il termine **Homset**

$$\forall x \in A.id(x) = x$$

rappresenta il morfismo id_A. Difatti, è banale verificare che, date $f : B \to A$,$g : A \to B$, si ha

$$(id \circ f)(x) = id(f(x)) = f(x)$$
$$(g \circ id)(x) = g(id(x)) = g(x)$$

come richiesto dalle equazioni 1.1.2 e 1.1.3. Infine, anche l'associatività della composizione di funzioni è banale da dimostrare. Ciò dimostra che **Set** è effettivamente una categoria.

Un altro esempio può essere dato dalla categoria dei monoidi **MON**. In questo caso, un oggetto corrisponde a un monoide, mentre un morfismo è dato da un omomorfismo di monoidi. Siano $(M,1_M,*_M)$,$(N,1_N,*_N)$ due monoidi. Ancora una volta, l'operazione di composizione dei morfismi corrisponderà alla composizione di funzioni (essendo gli omomorfismi tra gruppi effettivamente delle funzioni). La funzione $id(x) = x$ è difatti un omomorfismo da M in M, e quindi consiste nel morfismo identità id_M. Inoltre, dati due omomorfismi $f : M \to N$,$g : N \to L$, si ha

$$g(f(1_M)) = g(1_N) = 1_L$$
$$g(f(x *_M y)) = g(f(x) *_N f(y)) = g(f(x)) *_L g(f(y))$$

Ciò mostra che il risultato della composizione di due omomorfismi tra monoidi è anch'esso un omomorfismo. Avendo lasciato invariato l'operatore di composizione rispetto a **SET**, l'associatività è nuovamente banale da verificare. Tale ragionamento può essere ripetuto con i gruppi, ottenendo così la categoria **GRP**, e in generale con ogni algebra avente segnatura Ω.
Come ultimo esempio, si consideri un particolare monoide $(M,1,*)$. Si costruisca quindi una categoria $\mathcal{C}$ definendo un unico oggetto o, e scegliendo come morfismi gli elementi di tale monoide. Ovviamente, tali morfismi avranno l'oggetto o come dominio e codominio. L'operazione di composizione $\circ$ sarà data da $*$. Ponendo $id_o = 1$, si ottiene

$$x \circ id_o = x * 1 = x = 1 * x = id_o \circ x$$

Infine, $*$ è associativa per definizione stessa di monoide. Questo esempio mostra che qualsiasi monoide può essere rappresentato utilizzando le categorie.
L'ultimo esempio di categoria che si vuole presentare è dato da **CPO**: tale categoria verrà utilizzata più volte nel corso della tesi.
Dato un insieme C sul quale è definita una relazione d'ordine parziale $\sqsubseteq$, esso è detto essere **Completo** (**Complete Partially Ordered Set**, o **Cpo**) se, comunque si scelga una catena $c_0 \sqsubseteq c_1 \sqsubseteq cdots$, esiste il **least upper bound** di tale catena in C, ovvero un elemento $\bigsqcup_n c_n$ tale che:

$$\forall n.c_n \sqsubseteq \bigsqcup_n c_n$$
$$\exists c'.\forall n.c_n \sqsubseteq c' \quad \Rightarrow \quad \bigsqcup_n c_n \sqsubseteq c'$$

Data una funzione $f : (C_1, \sqsubseteq_1) \to (C_2, \sqsubseteq_2)$, dove C_1 e C_2 sono due Cpo, essa è detta essere continua se preserva l'ordinamento e i least upperbound, ovvero:

$$\begin{aligned} c \sqsubseteq_1 c' &\Rightarrow f(c) \sqsubseteq_2 f(c') \\ f(\sqcup_n^1 c_n) &= \sqcup_n^2 f(c_n) \end{aligned}$$

La categoria **CPO** ha come oggetti i Cpo, e come morfismi le funzioni continue tra Cpo: è immediato infatti verificare che la composizione di due funzioni continue è essa stessa una funzione continua, e che la funzione identità $id(c) = c$ è continua.

1.1.2 Rappresentazione tramite diagrammi

Data una categoria, è possibile rappresentarne gli oggetti e i morfismi utilizzando dei diagrammi. Tale rappresentazione gioca un ruolo importante per la verifica delle proprietà degli oggetti (o dei morfismi) di una categoria.

La rappresentazione tramite diagrammi consiste nel rappresentare graficamente un morfismo $f : A \to B$ come una freccia direzionata dall'oggetto A all'oggetto B. Un tipico esempio di diagramma è illustrato nella figura 1.1

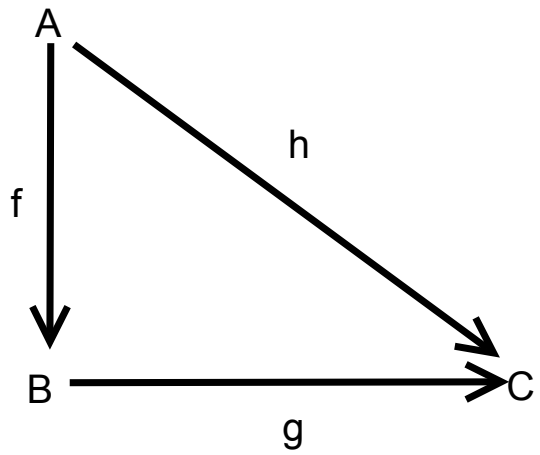
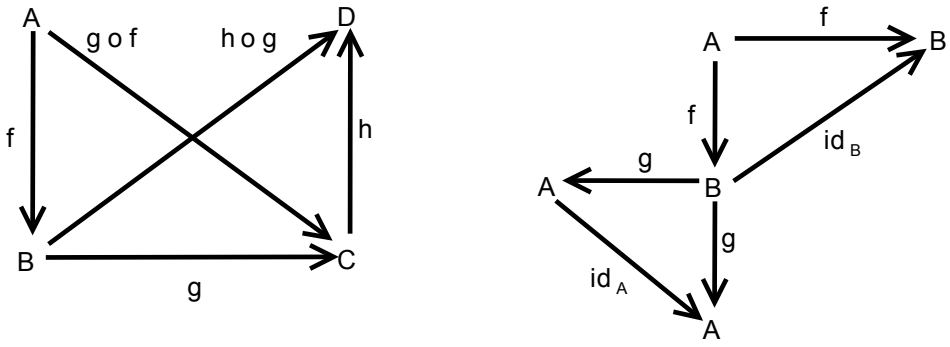


Figura 1.1: Esempio di diagramma

Dato un diagramma, si dirà che esso *commuta* se, e solo se, comunque si scelgano due punti rappresentanti rispettivamente due oggetti A e B, ogni percorso dal primo al secondo punto consiste di morfismi (derivati eventualmente dalla composizione di più morfismi) tra di loro uguali. Nell'esempio della figura 1.1, il diagramma mostrato commuta se e solo se $h = g \circ f$.

Le proprieta 1.1.1, 1.1.2 e 1.1.3 possono essere rappresentate in termini di diagrammi commutativi, mostrati in figura 1.2



(a) Proprietà Associativa

(b) Esistenza dell'Identità

Figura 1.2

In seguito si farà largo uso dei diagrammi commutativi.

1.2 Definizioni di base

Per poter analizzare le proprietà degli oggetti e dei morfismi presenti all'interno di una categoria, è necessario introdurre alcune definizioni basilari. In questa sezione verranno definite delle particolari classi di morfismi e di oggetti che soddisfano determinate proprietà.

1.2.1 Monomorfismi, Epimorfismi, Isomorfismi

Alcune delle più semplici proprietà che un morfismo può avere, all'interno di una categoria, è quella di essere un monomorfismo, un epimorfismo, oppure un isomorfismo. Queste giocano un ruolo importante nella dimostrazione di vari teoremi.

Definizione 1.2.1.

Monomorfismo

Data una categoria $\mathcal{C}$, un morfismo $f \in \mathcal{C}[B,C]$ è detto ***monomorfismo*** *se, e solo se, scelti arbitrariamente $g,h \in \mathcal{C}[A,B]$ vale l'implicazione*

$$f \circ g = f \circ h \Rightarrow h = g \tag{1.2.1}$$

Esempio 1.2.1.

Nella categoria ***Set****, i monomorfismi sono esattamente le funzioni iniettive.*
Difatti, sia $f : B \to C$ *una funzione iniettiva, e siano* $g : A \to B$,$h : A \to B$*. Per definizione di* f*, si ha che se* $f(g(x)) = f(h(x))$*, allora* $g(x) = h(x)$*.*
Diversamente, si supponga che f *non sia iniettiva. Dunque esiste* x,$y \in B : x \neq y$,$f(x) = f(y)$*. In questo caso si scelga* $g : \forall a \in A g(a) = x$*,* $h : \forall a \in A h(a) = y$*. Chiaramente si ha* $f(g(a)) = f(x) = f(y) = f(g(y))$*; tuttavia* $f \neq g$*, pertanto* f *non è un monomorfismo.*

♦

La definizione di epimorfismo segue dalla definizione 1.2.1. Difatti, la definizione di epimorfismo si ottiene dualmente a quella di monomorfismo.

Definizione 1.2.2.

Epimorfismo

Data una categoria $\mathcal{C}$*, un morfismo* $f \in \mathcal{C}[A,B]$ *è detto* ***epimorfismo*** *se, e solo se, scelti arbitrariamente* g,$h \in \mathcal{C}[B,A]$ *vale l'implicazione*

$$g \circ f = h \circ f \Rightarrow h = g \tag{1.2.2}$$

Esempio 1.2.2.

Nella categoria ***Set*** *gli epimorfismi sono esattamente le funzioni suriettive.*
Sia $f : A \to B$ *una funzione suriettiva, e siano* g,$h : B \to C$ *due funzioni tali che* $g(f(x)) = h(f(x))$*. Essendo* f *suriettiva, segue immediatamente il risultato* $g = h$*.*
Differentemente, sia f *una funzione non suriettiva. Esiste pertanto* $b \in B : f(x) \neq B$*. si costruiscano* g,h *in maniera tale che* $g(b) \neq h(b)$,$\forall x \in B \backslash \{b\}.g(x) = h(x)$*; l'uguaglianza* $g(f(x)) = h(f(x))$ *è ovviamente valida, tuttavia* $h \neq g$*. Dunque* f *non è un epimorfismo*

♦

Un ultima definizione necessaria è quella di isomorfismo. A livello informale, un isomorfimo è un morfismo che può essere invertito: la definizione formale, qui di sotto riportata, è facilmente ottenibile:

Definizione 1.2.3.

Isomorfismo

Data una categoria $\mathcal{C}$, un morfismo $f \in \mathcal{C}[A,B]$ è chiamato isomorfismo se, e solo se, esiste $f^{-1} \in \mathcal{C}[B,A]$ tale che

$$f^{-1} \circ f = id_A \tag{1.2.3}$$
$$f \circ f^{-1} = id_B \tag{1.2.4}$$

E' immediato notare che, in **Set**, gli isomorfismi sono tutte e sole le biezioni.
Due oggetti A,B sono isomorfi ($A \simeq B$) se, e solo se, esiste un isomorfismo da A in B (o, equivalentemente, da B in A). Per concludere l'esempio, in base a questa definizione due insiemi A,B sono isomorfi in **Set** se, e solo se, $|A| = |B|$.

1.2.2 Oggetti iniziali e terminali

Dopo aver definito alcune classi di morfismi, si vogliono definire ora alcuni particolari oggetti che è possibile incontrare all'interno di una categoria.
Un **oggetto iniziale** è un oggetto 0 per cui, dato un qualsiasi oggetto A, è sempre possibile trovare un morfismo $!_A : 0 \to A$. Inoltre, tale morfismo deve essere unico. Come risultato, si considerino due oggetti A,B all'interno di una categoria che possiede un oggetto iniziale 0. Siano inoltre $!_A : 0 \to A$ e $!_B : 0 \to B$ gli unici morfismi rispettivamente in $\mathcal{C}[0,A]$,$\mathcal{C}[0,B]$. Dato un qualsiasi morfismo $f : A \to B$, si ottiene

$$f \circ !_A = !_B$$

Definizione 1.2.4.

Oggetto Iniziale

*In una categoria $\mathcal{C}$ un oggetto 0 è detto **iniziale** se, e solo se, comunque si scelga $A \in Obj_{\mathcal{C}}$ esiste un unico morfismo $!_A : 0 \to A$.*

Dualmente è possibile dare la definizione di **oggetto terminale**.

Definizione 1.2.5.

Oggetto Terminale

In una categoria $\mathcal{C}$ un oggetto 1 è detto ***terminale*** *se, e solo se, comunque si scelga $A \in Obj_{\mathcal{C}}$ esiste un unico morfismo $!_A : A \to 1$.*

Esempio 1.2.3.

La categoria ***CPO*** *contiene un oggetto terminale 1, corrispondente al Cpo $(\{*\}, \sqsubseteq)$, dove $* \sqsubseteq *$. Difatti, scelto un Cpo $(C, \sqsubseteq_C)$, esiste un'unica funzione $!_C : C \to 1$, ovvero $!_C(c) = *$: è banale verificare che tale funzione è continua, e pertanto è un morfismo in* ***CPO****.*

♦

Per gli oggetti iniziali e terminali valgono le seguenti proprietà:

Proposizione 1.2.1. *In una categoria $\mathcal{C}$ l'oggetto iniziale (terminale) è unico a meno di isomorfismi.*

Dimostrazione. Verrà mostrato il risultato solamente per un oggetto iniziale. L'unicità (a meno di isomorfismi) dell'oggetto terminale segue difatti dalla dualità delle definizioni 1.2.4 e 1.2.5.

Siano 0_1,0_2 due oggetti iniziali in una categoria $\mathcal{C}$: si mostrerà che essi sono isomorfi.
Per definizione, esiste un unico morfismo $f : 0_1 \to 0_2$, ed esiste un unico morfismo $g : 0_2 \to 0_1$. Inoltre, sempre per definizione di oggetto iniziale, gli insiemi $\mathcal{C}[0_1,0_1]$ e $\mathcal{C}[0_2,0_2]$ sono costituiti rispettivamente dai soli morfismi id_{0_1},id_{0_2}. Allora valgono banalmente le equazioni

$$\begin{aligned} g \circ f &= id_{0_1} \\ f \circ g &= id_{0_2} \end{aligned}$$

Dunque $0_1 \simeq 0_2$. □

Data una categoria $\mathcal{C}$ con un oggetto terminale 1, un morfismo $1 \to A$ viene detto **Elemento globale** di A.

1.3 Costruzioni di base

La teoria delle categorie è ricca di costruzioni che permettono di verificare le proprietà delle strutture che si stanno rappresentando. Tuttavia, risulterebbe eccessivamente oneroso e di poca utilità elencare tutte le costruzioni esistenti. Si focalizzerà invece l'attenzione solamente sulle costruzioni basilari di cui si avrà bisogno in seguito. Per un qualsiasi approfondimento, si rimanda il lettore a [AL91], [Pie91], [Lan98].

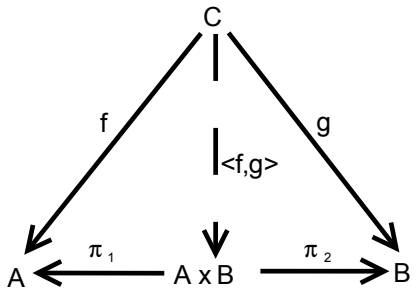
1.3.1 Prodotti e Coprodotti

Un prodotto di due oggetti A,B consiste di un oggetto $A \times B$, equipaggiato con due morfismi $\pi_1 : A \times B \to A$ e $\pi_2 : A \times B \to B$ la cui funzione è quella di estrarre l'informazione relativa ai due oggetti di cui è stato effettuato il prodotto. Tali morfismi, comunemente detti **proiettori**, devono preservare l'informazione relativa alle singoli componenti del prodotto; ciò è esprimibile all'interno della categoria asserendo che, dato un qualunque oggetto C e due morfismi $f : C \to A$, $g : C \to B$, esiste un unico morfismo $h : C \to A \times B$ tale che $\pi_1 \circ h = f$,$\pi_2 \circ h = g$. La seguente definizione stabilisce formalmente come viene costruito il prodotto di due oggetti:

Definizione 1.3.1.

Prodotto di oggetti

Data una categoria $\mathcal{C}$ e due suoi oggetti A,B, il loro prodotto $A \times B$ è un oggetto di $\mathcal{C}$, equipaggiato con due morfismi $\pi_1 : A \times B \to A$,$\pi_2 : A \times B \to B$, tali che, per ogni oggetto C e per ogni coppia di morfismi $f : C \to A$, $g : C \to B$, esiste un unico morfismo $< f,g >: C \to A \times B$ che fa commutare il seguente diagramma:



E' possibile provare il seguente risultato, di cui non verrà data la dimostrazione, sugli oggetti prodotto:

Proposizione 1.3.1. *In una categoria $\mathcal{C}$ il prodotto $A \times B$ di due oggetti A,B è unico a meno di isomorfismi.*

Una categoria in cui per ogni coppia di oggetti A,B esiste il prodotto $A \times B$ prende il nome di **Categoria Cartesiana** (**CC**).
La costruzione duale che si ottiene dal prodotto è detta **coprodotto**, e può essere ottenuta invertendo i morfismi nel diagramma mostrato in 1.3.1. Tale costruzione viene mostrata in figura 1.3

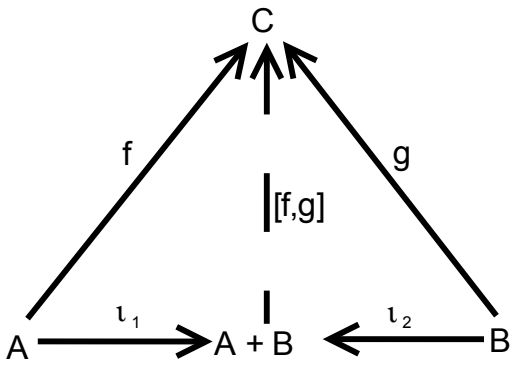


Figura 1.3: Coprodotto

In questo caso i morfismi ι_1 e ι_2 prendono il nome di iniettori. Il morfismo $[f,g]$ mostrato in figura 1.3 viene spesso indicato utilizzando la seguente sintassi:

$$\begin{aligned}[f,g](x) \quad = \quad & case\ x\ of \\ & \iota_1(a).f(a) \\ & \iota_2(b).g(b)\end{aligned}$$

I vincoli di commutatività del coprodotto di due oggetti può essere espresso in termini di equazioni, utilizzando la sintassi introdotta: si ottiene

$$\begin{aligned} & case\ \iota_1(a')\ of \\ & \quad \iota_1(a)\ f(a') \\ & \quad \iota_2(b).g(b) \quad = \quad f(a') \end{aligned} \tag{1.3.1}$$

$$\begin{aligned} & case\ \iota_2(b')\ of \\ & \quad \iota_1(a)\ f(a) \\ & \quad \iota_2(b).g(b) \quad = \quad g(b') \end{aligned} \tag{1.3.2}$$

$$\begin{aligned} & case\ x\ of \\ & \iota_1(a) = f(\iota_1(a)) \\ & \iota_2(b) = f(\iota_2(b) \quad = \quad f(x) \end{aligned} \tag{1.3.3}$$

Le equazioni 1.3.1 e 1.3.2 equivalgono alla commutatività dei due triangoli interni del diagramma 1.3. L'equazione 1.3.3 corrisponde invece all'unicità del morfismo $[f,g]$.
Dualmente alla proposizione 1.3.1, per i coprodotti si ottiene il seguente:

Proposizione 1.3.2. *In una categoria $\mathcal{C}$ il coprodotto $A + B$ di due oggetti A,B è unico a meno di isomorfismi.*

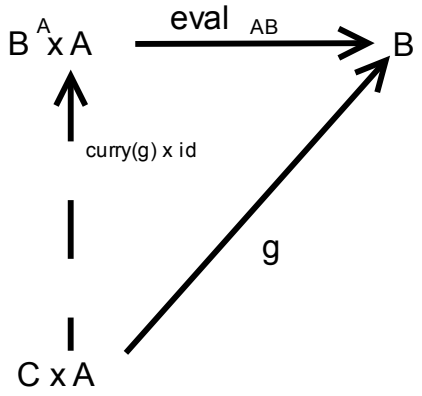
1.3.2 Esponenziazione

L'esponenziazione, vista da un punto di vista computazionale, rappresenta l'interpretazione categorica della **curryficazione**, cioè permette di rappresentare un morfismo da un prodotto di oggetti $A \times B$ in un oggetto C come un morfismo dall'oggetto A ad un oggetto C^B che rappresenta un morfismo avente dominio B e codominio C. All'interno della categoria **Set**, l'esponenziazione corrisponde alla trasformazione di una funzione di due argomenti, $f : A \times B \to C$ in una funzione $g : A \to C^B$, dove C^B è l'insieme $\{f : B \to C\}$.

Definizione 1.3.2.

Esponenziazione

*In una categoria $\mathcal{C}$ un oggetto B^A è detto **esponenziale** se esiste un morfismo $eval_{AB} : (B^A \times A) \to B$ tale che, per ogni oggetto C ed ogni morfismo $g : (C \times A) \to B$ esiste un unico morfismo $curry(g) : C \times B^A$ che fa commutare il seguente diagramma:*



Una categoria in cui per ogni coppia di oggetti A,B esistono gli oggetti $A \times B$ e B^A, e per la quale esiste un oggetto terminale, prende il nome di **categoria cartesiana chiusa** (**CCC**).

1.3.3 Natural Number Objects

Nello studio dei linguaggi di programmazione tramite l'utilizzo della teoria delle categorie, è spesso necessario interpretare l'insieme dei numeri naturali come un oggetto di una categoria. Tale oggetto dovrà possedere dei morfismi e delle proprietà che permettano di soddisfare gli assiomi di Peano, tipici della struttura dei numeri naturali. In particolare, si devono esplicitare tramite morfismi la costante **zero** e la funzione successore s.
In una categoria $\mathcal{C}$ con oggetto terminale 1, l'oggetto che viene utilizzato per interpretare l'insieme dei numeri naturali prende il nome di **Natural Number Object**.

Definizione 1.3.3.

Natural Number Object

*Data una categoria $\mathcal{C}$ con oggetto terminale 1, un oggetto N avente un elemento globale $z : 1 \to N$ e un endomorfismo $s : N \to N$ è detto essere un **Natural Number Object (nno)** se, e solo se, per ogni oggetto A, un suo elemento globale $q : 1 \to A$ e un suo endomorfismo $f : A \to A$, esiste un unico morfismo $u : N \to A$ tale che*

$$u \circ z = q \tag{1.3.4}$$
$$u \circ s = f \circ u \tag{1.3.5}$$

Dato un **nno** N in una categoria $\mathcal{C}$, e un morfismo $u : N \to A$, questi può essere espresso ricorsivamente in funzione dell'elemento globale $q : 1 \to A$ e dell'endomorfismo $f : A \to A$. Se si considera la categoria **Set** (dotata di oggetto terminale 1 e di un nno N), una funzione $u : N \to A$ può essere espressa come:

$$\begin{aligned} u(z) &= q \\ \forall y \in N. u(s(y)) &= f(u(y)) \end{aligned}$$

1.4 Funtori

Sebbene lo studio di una singola categoria possa portare a scoprire proprietà interessanti, spesso si è interessati a cercare una relazione che intercorre tra due diverse categorie.

Gli strumenti matematici che permettono di mappare una categoria in un'altra prendono il nome di **Funtori**. Un funtore $F : \mathcal{C} \to \mathcal{D}$ è costituito da una coppia di funzioni, $F_{obj} : Obj_{\mathcal{C}} \to Obj_{\mathcal{D}}$ e $F_{mor} : Mor_{\mathcal{C}} \to Mor_{\mathcal{D}}$, tali che la struttura della categoria $\mathcal{C}$ venga preservata all'interno della categoria $\mathcal{D}$.

Definizione 1.4.1.

Funtore Covariante

Date due categorie $\mathcal{C}$,$\mathcal{D}$, un ***Funtore (Covariante)*** $F : \mathcal{C} \to \mathcal{D}$ *è una coppia di funzioni* $F_{obj} : Obj_{\mathcal{C}} \to Obj_{\mathcal{D}}$,$F_{mor} : Mor_{\mathcal{C}} \to Mor_{\mathcal{D}}$ *tali che:*

- $f : A \to B \Rightarrow F_{mor}(f) : F_{obj}(A) \to F_{obj}(B)$.
- $F_{mor}(id_A) = id_{F_{obj}(id_A)}$.
- $F_mor(g \circ_{\mathcal{C}} f) = F_{mor}(g) \circ_{\mathcal{D}} F_{mor}(f)$.

Per non appesantire troppo la notazione, si utilizzerà la notazione F in entrambi i casi in cui ci si stia riferendo alla funzione F_{mor} o F_{obj}; inoltre, le parentesi saranno omesse là dove non sono strettamente necessarie.

In seguito, verranno spesso considerati dei funtori del tipo $F : \mathcal{C} \to \mathcal{C}$. In letteratura ci si riferisce a tali funtori con il termine di **Endofuntori**.

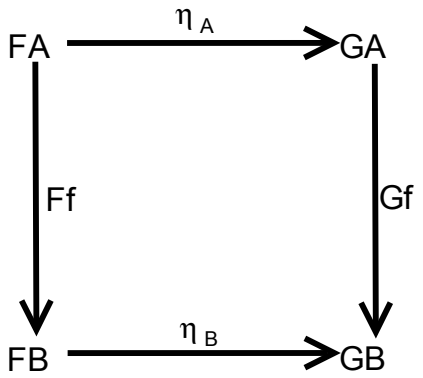
1.5 Trasformazioni Naturali

La definizione 1.4.1 è stata introdotta per formalizzare una associazione da una categoria ad un altra. Effettuando un ulteriore passo di astrazione, si vogliono introdurre degli strumenti che mettano in relazione due funtori F,$G : \mathcal{C} \to \mathcal{D}$. In altre parole, si vuole introdurre uno strumento per mappare il funtore F nel funtore G, in maniera tale da preservare la struttura di F in G. Tali strumenti vengono formalizzati nel concetto di **trasformazione naturale**.

Definizione 1.5.1.

Trasformazione Naturale

Siano $\mathcal{C}$ e $\mathcal{D}$ due categorie, e siano F,G : $\mathcal{C} \to \mathcal{D}$ due funtori. Una ***trasformazione naturale*** *$\eta : F \to G$ è una funzione che assegna a ogni oggetto $A \in C_{obj}$ un morfismo $\eta_A : FA \to GA$ nella categoria $\mathcal{D}$ in maniera tale che, per ogni morfismo $f : A \to B$, il seguente diagramma commuta:*



Chiaramente, la funzione $id_F : F \to F$ è una trasformazione naturale, il cui effetto è quello di mappare il funtore F in sè stesso. Si può inoltre notare che la composizione di due trasformazioni naturali $\eta : F \to G$ e $\mu : G \to H$ è a sua volta una trasformazione naturale, come si può notare dal diagramma in figura 1.4.
Data una trasformazione naturale $\eta : F \to G$, essa è detta essere un **isomorfismo naturale** se esiste una trasformazione naturale $\eta^{-1} : G \to F$ tale che $\eta^{-1}(\eta) = id_F$, $\eta(\eta^{-1}) = id_G$. Equivalentemente, η è un isomorfismo naturale se, e solo se, per ogni oggetto A si ha che η_A è un isomorfismo.
Sia $\eta : F \to G$ una trasformazione naturale, dove F e G sono funtori da $\mathcal{C}$ a $\mathcal{D}$. Sia inoltre $H : \mathcal{C} \to \mathcal{D}$ un funtore: vengono indotte le trasformazioni naturali η_H e $H\eta$, definite come segue:

$$(H\eta)_A = K\eta_A : (HF)A \to (HG)A \tag{1.5.1}$$
$$(\eta_H)_A = \eta_{HA} : (FH)A \to (GH)A \tag{1.5.2}$$

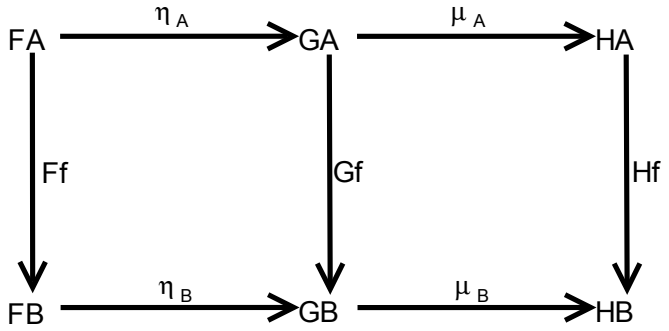


Figura 1.4: Composizione di trasformazioni naturali

1.6 Monadi, Triple di Kleisli

Presa in considerazione una categoria $\mathcal{C}$, non è difficile costruire una nuova categoria i cui oggetti siano gli endofuntori $F : \mathcal{C} \to \mathcal{C}$ e i cui morfismi siano le trasformazioni naturali. Per un endofuntore F, difatti, non è difficile mostrare che la trasformazione naturale id_F è il morfismo identità. Inoltre, l'associatività della composizione delle trasformazioni naturali è facilmente dimostrabile tramite diagrammi.

Si ottiene dunque una categoria **Endf** degli endofuntori. In tale categoria è inoltre presente un oggetto iniziale, dato dal funtore identità 1, dove $1A = A$, $1f = f$. Un monoide, in tale categoria, è dato da un funtore T e da due trasformazioni naturali $\eta : 1 \to T$ e $\mu : T^2 \to T$ tali che:

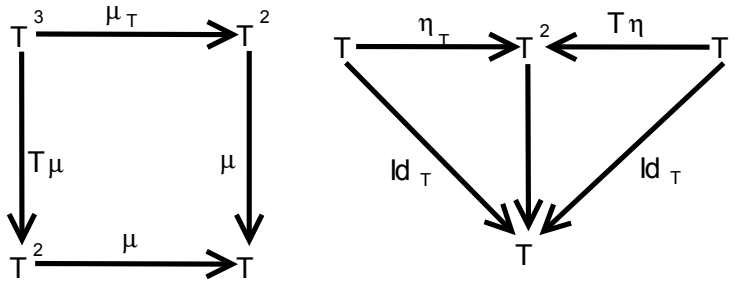
$$\mu\mu_T = \mu(T\mu) \tag{1.6.1}$$
$$\mu(\eta_T) = id_T \tag{1.6.2}$$
$$\mu(T\eta) = id_T \tag{1.6.3}$$

Un monoide nella categoria degli endofuntori prende il nome di **Monade**.

Definizione 1.6.1.

Monade

Una monade è una tripla $< T,\eta,\mu >$, dove T è un endofuntore ed $\eta : 1 \to T$,$\mu : T^2 \to T$ sono due trasformazioni naturali tali che i seguenti diagrammi commutano:



Esiste una seconda costruzione che si dimostra essere equivalente alla definizione 1.6.1. Data una funzione T tra gli oggetti di una categoria $\mathcal{C}$, una tripla di Kleisli è costituita dalla funzione T, da una famiglia di funzioni $\eta_A : A \to T$, e da un operatore di **lifting** $\_^*$ tale che, data $f : A \to TB$, allora $f^* : TA \to TB$, tali da soddisfarre le equazioni che vengono presentati nella definizione seguente:

Definizione 1.6.2.

Tripla di Kleisli

*Una **tripla di Kleisli** in una categoria $\mathcal{C}$ è una tripla $< T;\eta,\_^* >$, dove $T : Obj_{\mathcal{C}} \to Obj_{\mathcal{C}}$ è una associazione di oggetti, η è una famiglia di morfismi $\eta_A : A \to TA$ e $\_^*$ è un operatore tale che, se $f : A \to TB$, allora $f^* : TA \to TB$. Inoltre, valgono le seguenti equazioni:*

$$\eta_A^* = id_{TA} \tag{1.6.4}$$
$$\eta_A; f^* = f \tag{1.6.5}$$
$$f^*; g^* = (f; g^*)^* \tag{1.6.6}$$

Per concludere l'introduzione alla teoria delle categorie, verrà provato che ogni tripla di Kleisli è una monade, e viceversa:

Proposizione 1.6.1 ([Mog89a])**.** *Esiste una corrispondenza uno a uno tra triple di Kleisli e monadi.*

Dimostrazione. Data una tripla di Kleisli $< T,\eta,\_^* >$, la monade corrispondente è $< T,\eta,\mu >$, dove T è l'estensione della funzione T ad un endofuntore, ponendo $T(F : A \to B) = (f; \eta_B^*)$, e $\mu_A = id_{TA}^*$.
Differentemente, data una monade $< T,\eta,\mu >$, la corrispondente tripla di Kleisli è $< T,\eta,\_^* >$, dove T è la restrizione del funtore T agli oggetti, e $(f : A \to TB)^* = Tf; \mu_B$. □

L'ultimo strumento di cui si avrà bisogno in seguito è dato dalle **monadi forti**. Una monade forte su una **CC** $\mathcal{C}$ è una quadrupla $< T,\eta,\mu,t >$, dove T,η,μ costituiscono una monade, e t è una trasformazione naturale $t_{A,B} : A \times TB \to T(A \times B)$, tale che i diagrammi 1.5, 1.6, 1.7 commutano. In tali diagrammi r ed α sono degli isomorfismi naturali:

$$r_A : 1 \times A \to A$$
$$\alpha_{A,B,C} : (A \times B) \times C \to A \times (B \times C)$$

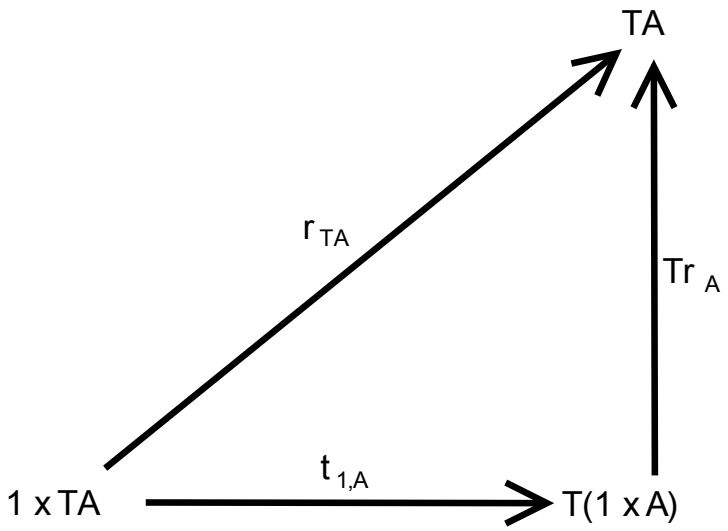


Figura 1.5: Diagramma 1 per le monadi forti

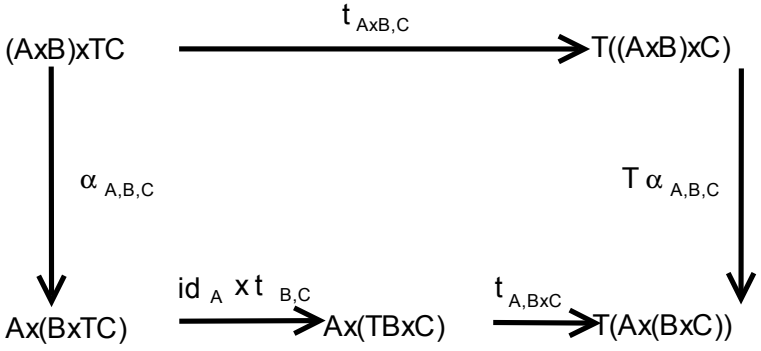


Figura 1.6: Diagramma 2 per le monadi forti

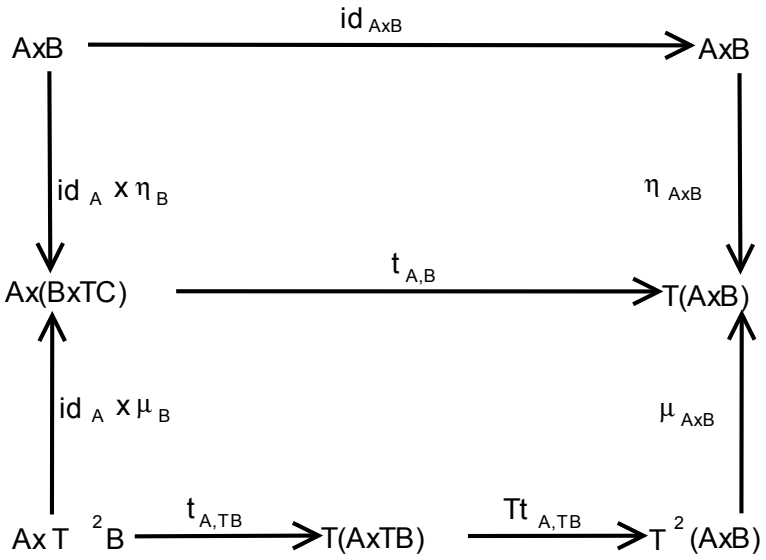


Figura 1.7: Diagramma 3 per le monadi forti

2 Semantica categoriale

La teoria delle categorie, introdotta nel capitolo 1, è stata utilizzata nell' informatica per sviluppare la semantica denotazionale per diversi linguaggi. Verrà esaminato un esempio di semantica, tratto da [Pie91], per l'interpretazione del λ-calcolo tipato con coppie all'interno di una **CCC**.

Sfruttando il concetto di monade, in [Mog89b] Eugenio Moggi ha introdotto il modello del **λ-calcolo computazionale** (λ_C). In questo modello le funzioni non vengono più considerate come associazioni tra coppie di valori, ma bensì come associazioni fra valori e computazioni, opportunamente rappresentate da una monade $< T,\eta,\mu >$. La possibilità di definire differenti effetti computazionali utilizzando monadi differenti permette di utilizzare il λ-calcolo computazionale come un framework per l'interpretazione semantica di linguaggi con differenti caratteristiche. Nel capitolo 4 verranno mostrati degli esempi concreti di come il λ-calcolo computazionale possa essere utilizzato per interpretare le transizioni che i programmi di un linguaggio imperativo operano sulle configurazioni di memoria.
Infine, allo scopo di introdurre un concetto di modularità, in [Mog89a] e [LHJ95] vengono introdotti i **costruttori di monadi**, strumenti che permettono, a partire da una monade data, di estenderla al fine di ottenere una nuova monade, la cui interpretazione in λ_C può essere ottenuta dalla monade originaria. Tali costruttori verranno tuttavia introdotti nel capitolo 3, quando verranno mostrati diversi effetti computazionali rappresentabili tramite monadi.

2.1 Semantica categorica del lambda calcolo tipato

2.1.1 Sintassi e sistema di tipi

Il λ-calcolo semplice tipato con coppie ($\lambda^{\rightarrow}_{\times}$) è uno dei più semplici modelli funzionali per la rappresentazione delle funzioni. Scelto un insieme di tipi costanti κ, è possibile costruire a partire da essi i **tipi freccia** e i **tipi prodotto**. Se τ_1, e τ_2 sono due tipi, allora lo saranno

anche il tipo freccia $\tau_1 \to \tau_2$ e il tipo prodotto $\tau_1 \times \tau_2$. Si suppone, inoltre, di disporre di un tipo speciale $Unit$. Formalmente, i tipi del linguaggio sono definiti tramite la seguente BNF:

$$\tau \ ::= \ Unit \mid \kappa \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \tag{2.1.1}$$

Ad ogni termine del λ-modello che si andrà a definire verrà associato un tipo. Oltre ad utilizzare le ben note operazioni di astrazione $\lambda x.$ e di composizione MN, vengono introdotti il prodotto di termini $< M,N >$ e gli operatori di proiezione $\pi_1 M$, e $\pi_2 M$. Ovviamente, gli operatori di proiezione potranno essere applicati solamente a un termine il cui tipo sia un tipo prodotto. Infine, si assumerà che il linguaggio sia equipaggiato con una costante $unit : Unit$
La BNF risultante per tale linguaggio è la seguente:

$$M \ ::= \ unit \mid x \mid \lambda x.M \mid (M_1 M_2) \mid \ < M_1,M_2 > \ \mid \pi_1 M \mid \pi_2 M \tag{2.1.2}$$

Infine, così come avviene per ogni λ-calcolo tipato, è necessario introdurre un sistema di tipi per la rappresentazione dei termini validi all'interno del linguaggio. Non è difficile ricavare un sistema di tipi per il linguaggio preso in considerazione

Sistema di Tipi 2.1.1. [1]

$$\frac{}{\Gamma,x : A \vdash x : A} \qquad \frac{}{\vdash unit : Unit}$$

$$\frac{\Gamma,x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B} \qquad \frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$$

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B}{\Gamma \vdash < M,N > : A \times B}$$

$$\frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \pi_1 M : A} \qquad \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \pi_2 M : B}$$

2.1.2 Semantica Categorica

Dato una λ-modello, è possibile definirne una semantica categorica interpretandone i tipi e i termini all'interno di una **CCC**. In particolare, una funzione di interpretazione $[\![\cdot]\!]$ associerà ad ogni tipo un oggetto della categoria presa in considerazione, mentre ad ogni giudizio ottenuto $\Gamma \vdash M : A$ verrà associato un morfismo all'interno della stessa **CCC**.

L'idea alla base dell'interpretazione di un λ-modello all'interno di una CCC consiste nel definire l'interpretazione dei tipi, e quindi da essa stabilire come vengano interpretati i contesti Γ e i giudizi $\Gamma \to M$. Tipicamente, una volta definita l'interpretazione di un contesto

[1] *Per non appesantire troppo il sistema di tipi, sono state omesse alcune regole quali il weakening: per il sistema di tipi completo, si faccia riferimento a [AL91]*

$[\![\Gamma]\!]$, l'interpretazione di un giudizio corrisponderà a un morfismo $[\![\Gamma \vdash M : A]\!] : [\![\Gamma]\!] \to [\![A]\!]$.

Nel sistema $\lambda^{\to}_{\times}$ introdotto, ad ogni tipo di base $A \in \kappa$ viene associato un oggetto $A_{\mathcal{C}}$ della **CCC** $\mathcal{C}$ scelta per l'interpretazione. Inoltre, il tipo $Unit$ verrà interpretato nell'oggetto terminale di tale **CCC**. Come è possibile intuire, ai tipi freccia verrà assegnato l'oggetto corrispondente all'esponenziazione dei due tipi che la compongono, mentre l'interpretazione di un tipo prodotto corrisponderà al prodotto dell'interpretazione dei singoli tipi che lo compongono: si ottiene la seguente definizione induttiva per l'interpretazione dei tipi in $\mathcal{C}$:

$$\begin{aligned} [\![A]\!] &= A_{\mathcal{C}} \ (A \in \kappa) \\ [\![Unit]\!] &= 1_{\mathcal{C}} \\ [\![A \times B]\!] &= [\![A]\!] \times [\![B]\!] \\ [\![A \to B]\!] &= [\![B]\!]^{[\![A]\!]} \end{aligned}$$

L'interpretazione dei contesti consiste nell'oggetto ottenuto dal prodotto dell'interpretazione dei tipi delle singole componenti che lo compongono. Anche in questo caso l'interpretazione è definibile induttivamente, associando l'oggetto terminale della **CCC** al contesto vuoto:

$$\begin{aligned} [\![\emptyset]\!] &= 1_{\mathcal{C}} \\ [\![\Gamma,x : A]\!] &= [\![\Gamma]\!] \times [\![A]\!] \end{aligned}$$

Infine, le equazioni di seguito riportate permettono di interpretare i giudizi come morfismi all'interno di $\mathcal{C}$. L'idea basilare, utilizzata spesso nella semantica categorica, è quella di utilizzare l'operatore *curry* per interpretare l'astrazione di un termine, e l'operatore *eval* per interpretare l'applicazione di due termini. Nelle equazioni che verranno presentate, ci si riferirà a π_1 e π_2 sia quali morfismi di un prodotto in $\mathcal{C}$, sia come oggetti sintattici della λ-teoria presentata. Sarà comunque possibile distinguere dal contesto il modo in cui π_1 e π_2 vengono utilizzati:

$$\begin{aligned} [\![\Gamma \vdash unit : Unit]\!] &= !_{[\![\Gamma]\!]} \\ [\![\Gamma \vdash c : B]\!] &= c \circ ! \\ [\![\Gamma,x : A \vdash x : A]\!] &= \pi_2 \\ [\![\Gamma,x : A \vdash x' : A']\!] &= [\![\Gamma \vdash x' : A']\!] \circ \pi_1, x' \neq x \\ [\![\Gamma \vdash \lambda x.M : A \to B]\!] &= curry([\![\Gamma,x : A \vdash M : B]\!]) \\ [\![\Gamma \vdash MN : B]\!] &= eval_{CB} \circ < [\![\Gamma \vdash ,M : C \to B]\!], [\![\Gamma \vdash N : C]\!] > \\ [\![\Gamma \vdash < M,M' >: B \times B']\!] &= < [\![\Gamma \vdash M : B]\!], [\![\Gamma \vdash M' : B']\!] > \\ [\![\Gamma \vdash \pi_1 M : B]\!] &= \pi_1 \circ [\![\Gamma \vdash M : B \times B']\!] \\ [\![\Gamma \vdash \pi_2 M : B']\!] &= \pi_2 \circ [\![\Gamma \vdash M : B \times B']\!] \end{aligned}$$

Questo esempio mostra come sia possibile interpretare un λ-modello all'interno di una **CCC**: viceversa, data una categoria $\mathcal{C}$, è possibile definire un λ-modello a partire da

essa. Tuttavia, tale procedimento esula dagli scopi prefissi in questa tesi, pertanto non si approfondirà l'argomento. Maggiori informazioni possono essere trovate in [Pie91] e [AL91].

2.2 Lambda calcolo computazionale

In [Mog89b], Eugenio Moggi definisce un metalinguaggio la cui interpretazione permette di esprimere diversi effetti computazionali. Nel suo lavoro, una **computazione** viene definita formalmente come una tripla di Kleisli $< T,\eta,\_^* >$ (o, equivalentemente, da una monade $< T,\eta,\mu >$). Sfruttando tale definizione, le funzioni di un linguaggio di programmazione non vengono più definite come associazioni. Nel suo articolo, Moggi definisce un metalinguaggio, che prende il nome di **λ-calcolo computazionale**, dotato di un costruttore di tipi T: Dato un tipo A del metalinguaggio, TA rappresenta il tipo delle computazioni di A.

Considerata una monade $< T,\eta,\mu >$, si consideri la corrispondente tripla di Kleisli $< T,\eta,\_^* >$. Una funzione che mappa un valore in una computazione è un morfismo $f : A \to TB$. La composizione delle funzioni rappresentata dai due morfismi $f : A :\to TB$ e $g : B \to TC$ può essere ottenuta utilizzando l'operatore di lifting $\_^*$: si ha infatti $f; g^* : A \to TC$. Inoltre, è possibile trasformare un valore di tipo A in una computazione su tale tipo ricorrendo al morfismo $\eta_A : A \to TA$.
Le operazioni η e $\_^*$ vengono integrate all'interno del framework di Moggi utilizzando due particolari costrutti sintattici: *val* e *let*. Prima di presentare come tali costrutti vengono utilizzati nel λ-calcolo computazionale, è necessario però definire i tipi utilizzati all'interno del linguaggio.
Dato un insieme di tipi di base κ, la seguente BNF definisce i tipi utilizzati all'interno di λ_C:

$$\tau \ ::= \ \kappa \mid T\tau \mid \tau_1 \to \tau_2 \tag{2.2.1}$$

Il costrutto *val* corrisponde alla funzione η della tripla di Kleisli presa in considerazione. Se un termine M ha tipo A, allora $valM : TA$. Il costrutto *let*, invece, corrisponde alla composizione dei termini. Come mostrato in precedenza, se l'interpretazione dei termini M,N corrisponde ai morfismi f,g, la composizione di tali termini verrà interpretata nel morfismo $f; g^*$. Tale composizione di termini è esprimibile nel λ-calcolo computazionale come let $x = M$ in N. L'insieme dei termini di λ_C viene così definito:

$$M \ ::= \ x \mid val\ M \mid let\ x = M\ in\ N \tag{2.2.2}$$

E' possibile adesso presentare il sistema di tipi di λ_C: le uniche due regole di tipaggio contenute al suo interno sono le seguenti:

Sistema di Tipi 2.2.1.

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash valM : TA} \qquad \frac{\Gamma \vdash M : TA \qquad \Gamma,x : A \vdash N : TB}{\Gamma \vdash let\ x = M\ in\ N : TB}$$

E' possibile estendere il sistema λ_C in modo da incorporare anche gli operatori di astrazione ed applicazione. In seguito, quando sarà necessario, si assumerà implicitamente tale sistema come ambiente di riferimento.

2.2.1 Assiomi del lambda calcolo computazionale

Nella sezione 1.6 sono state introdotte le triple di Kleisli e le equazioni che esse devono soddisfare, 1.6.4, 1.6.5 e 1.6.6. In questa sezione è poi stato introdotto il λ-calcolo computazionale, introducendo gli operatori *let* e *val*. Le equazioni delle triple di Kleisli, opportunamente riscritte tramite gli operatori *let* e *val*, definiscono una teoria equazionale per il λ-calcolo computazionale.

Data una tripla di Kleisli $< T,\eta,\text{-}^* >$, L'equazione 1.6.4 stabilisce che $eta^*_A = id_{TA}$. Il termine id_{TA} è facilmente rappresentabile come $\lambda x : TA.x$. Per quanto riguarda la funzione η^*_A, si consideri che η_A viene rappresentata in λ_C come $val\ x$, dove $x : A$. Il passaggio di un valore alla funzione $valx$ può essere rappresentato usando il costrutto let: si ottiene dunque il termine $let\ x = M\ in\ val\ x$. Si noti che in questo termine viene passato esplicitamente il termine M come elemento di input di $val\ x$. Pertanto, tale valore dovrà essere utilizzato come argomento di input del termine associato al morfismo id_{TA}, ottenendo pertanto $(\lambda x : TA.x)M \rightarrow_\beta M$. L'equazione che si ottiene è pertanto la seguente:

$$let\ x = M\ in\ val\ x = M \tag{2.2.3}$$

L'equazione 1.6.5 corrisponde a $\eta_A; f^* = f$. Si è già notato che, date due funzioni $f : A \rightarrow TB$,$g : B \rightarrow TC$, allora $f; g^*$ è esprimibile in λ_C come $let\ x = M\ in\ N$, dove M rappresenta il morfismo f e N rappresenta il morfismo g. Si è inoltre mostrato che il morfismo η_A è rappresentabile tramite il termine $val\ A$. Dunque, il morfismo $\eta_A; f^*$ è rappresentabile in λ_C come:

$$let\ x = val\ M\ in\ N$$

Anche in questo caso viene esplicitamente passato un valore in input al termine N. Pertanto, tale passaggio dovrà apparire anche nel lato destro dell'equazione, che corrisponderà pertanto ad $N\{M/x\}$. Si ottiene dunque la seguente relazione fra termini:

$$let\ x = val\ M\ in\ N = N\{M/x\} \tag{2.2.4}$$

L'ultima equazione, 1.6.6, stabilisce che $(f; g^*)^* = f^*; g^*$. Se M rappresenta il morfismo $f : A \rightarrow TB$, ed N rappresenta il morfismo $g : B \rightarrow TC$, il lifting del morfismo $f; g^*$ può essere espresso in λ_C utilizzando l'operatore *let*, così da passare un termine $L : TA$ come argomento di input del termine rappresentante il morfismo $f; g^*$, dato da $let\ x = M\ in\ N$: il termine che si ottiene è dato da

$$let\ y = L\ in\ (let\ x = M\ in\ N)$$

L'equazione $f^*; g^*$ si può tradurre effettuando il lifting del termine M, e quindi passando la computazione ottenuta come risultato al lifting del termine N: nel linguaggio λ_C si ottiene

$$let\ x = \ (let\ y = L\ in\ M)\ in\ N$$

Occorre notare, in questo caso, che tale equazione risulta valida solamente nel caso in cui y non appartenga alle variabili libere di N. L'ultimo assioma che si ottiene per la teoria equazionale del λ-calcolo computazionale è allora:

$$let\ y = L\ in\ (let\ x = M\ in\ N)\ =\ let\ x = (let\ y = L\ in\ M)\ in\ N\ (y \notin FV(N)) \quad (2.2.5)$$

Gli assiomi 2.2.3, 2.2.4 e 2.2.5 possono essere utilizzati per mostrare che, data un'interpretazione dei costrutti *let* e *val* all'interno di una categoria $\mathcal{C}$, questi corrispondono effettivamente a una tripla di Kleisli (e quindi ad una monade).

2.2.2 Interpretazione dei termini

Così come nella sezione 2.1 si è mostrato come ricavare una semantica categoria per il λ-calcolo tipato con coppie, si vuole ora esibire una semantica categoria per il λ-calcolo computazionale. Fissata una **CCC** $\mathcal{C}$ e un endofuntore $T : \mathcal{C} \to \mathcal{C}$, l'interpretazione dei tipi corrisponderà nuovamente all'associazione di un oggetto ad un tipo.
Ad ogni tipo base $A \in \kappa$ l'interpretazione $[\![A]\!]$ farà corrispondere un oggetto $A_{\mathcal{C}}$. Per quanto riguarda un tipo della forma TA, la sua interpretazione dovrà dipendere da $[\![A]\!]$. Poiché in λ_C il tipo TA rappresenta una computazione su un valore di tipo A, e poiché nella categoria $\mathcal{C}$ fissata il mapping tra un tipo e una computazione su tale tipo viene rappresentata dal funtore T, si definisce $[\![TA]\!] = T[\![A]\!]$. Infine, per quanto riguarda i tipi freccia, si avrà nuovamente $[\![A \to B]\!] = [\![B]\!]^{[\![A]\!]}$. Recapitolando:

$$[\![A]\!] = A_{\mathcal{C}}\ (A \in \kappa) \quad (2.2.6)$$
$$[\![TA]\!] = T[\![A]\!] \quad (2.2.7)$$
$$[\![A \to B]\!] = [\![B]\!]^{[\![A]\!]} \quad (2.2.8)$$

L'interpretazione di un contesto è identica a quella fornita per il λ-calcolo tipato con coppie introdotto in sezione 2.1. Per completezza viene comunque riesibita qui di seguito:

$$[\![\emptyset]\!] = 1 \quad (2.2.9)$$
$$[\![\Gamma, x : A]\!] = [\![\Gamma]\!] \times [\![A]\!] \quad (2.2.10)$$

Infine, si deve definire l'interpretazione dei giudizi. Si sono già analizzati, per il λ-calcolo tipato con coppie, molti casi che ricorreranno nell'interpretazione dei giudizi di λ_C, come ad esempio $[\![\Gamma, x : A \vdash x : A]\!] = \pi_2$. Le uniche regole di tipaggio per cui devono essere definite le interpretazioni corrispondono a quelle introdotte nel sistema di tipi 2.2.1
Un giudizio del tipo $\Gamma \vdash val\ M : TA$ può essere ottenuto solamente dal giudizio $\Gamma \vdash M : A$, in virtù delle regole di inferenza presentate nel sistema di tipi 2.2.1 presentato. L'interpretazione per tale giudizio è data da un morfismo $[\![\Gamma \vdash M : A]\!] : [\![\Gamma]\!] \to [\![A]\!]$. Nelle

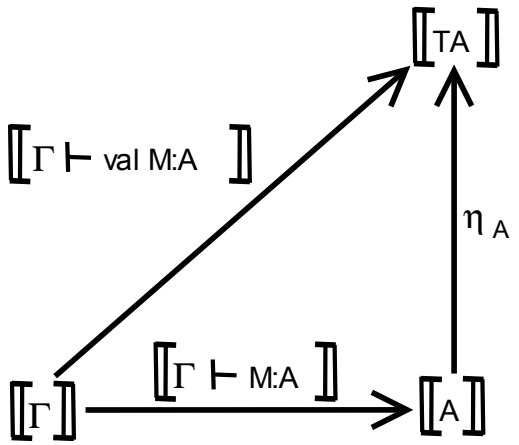


Figura 2.1: Interpretazione del val

sezioni precedenti, inoltre, si è già accennato che il costrutto $val\ M : A$ corrisponde al morfismo η_A della tripla di Kleisli $< T,\eta,\_^* >$ presa in considerazione. Il diagramma in figura 2.1
Mostra come ottenere l'interpretazione del giudizio $\Gamma \vdash val\ M : TA$, ricordando che $[\![TA]\!] = T[\![A]\!]$ da 2.2.7.

L'interpretazione per il *let* verrà analizzata in dettaglio: in precedenza si è già accennato che, se $\Gamma \vdash M : TA$ corrisponde ad una morfismo f, e $\Gamma,x : A \vdash N : TB$ corrisponde ad un morfismo g, allora $\Gamma \vdash let\ x = M\ in\ N : TB$ corrisponde a $f; g^*$. In termini di interpretazione, si ha allora:

$$\begin{aligned} [\![\Gamma \vdash M : TA]\!] &= f : [\![\Gamma]\!] \to T[\![A]\!] \\ ]\!]\Gamma,x : A \vdash N : TB]\!] &= g : [\![\Gamma]\!] \times [\![A]\!] \to T[\![B]\!] \end{aligned}$$

Dal secondo si ottiene immediatamente il morfismo

$$g^* = [\![\Gamma,x : A \vdash N : TB]\!]^* : T([\![\Gamma]\!] \times [\![A]\!]) \to T[\![B]\!]$$

Si noti che $codf \neq domg$, e pertanto la composizione $f; g^*$ non è valida. Ciò è vero in quanto non si è tenuto conto della presenza di un contesto. E' pertanto necessario, a partire dal morfismo $f : [\![\Gamma]\!] \to T[\![A]\!]$, ricavarne un secondo il cui codominio coincida con il dominio della funzione g. A questo scopo, si pone il vincolo di utilizzare una **monade forte**. In questo modo, è possibile costruire, partendo da f, il morfismo $id_{[\![TA]\!]} \times f : [\![\Gamma]\!] \times T[\![A]\!]$, e quindi applicare ad essi la forza $t_{\Gamma,A}$, ottenendo così un oggetto di tipo $T([\![\Gamma]\!] \times [\![A]\!])$ al quale può essere applicato il morfismo g^*. Il diagramma che ne risulta è mostrato in

figura 2.2, dove $s : [\![\Gamma \times T[\![A]\!] \to T([\![\Gamma]\!] \times [\![A]\!])$ rappresenta la forza della monade per gli oggetti $[\![\Gamma]\!]$ e $[\![A]\!]$. Dal diagramma della figura 2.2 si può allora identificare il morfismo $[\![\Gamma \vdash let\ x = M\ in\ N]\!]$ come $< f,id_A >; s; g^*$

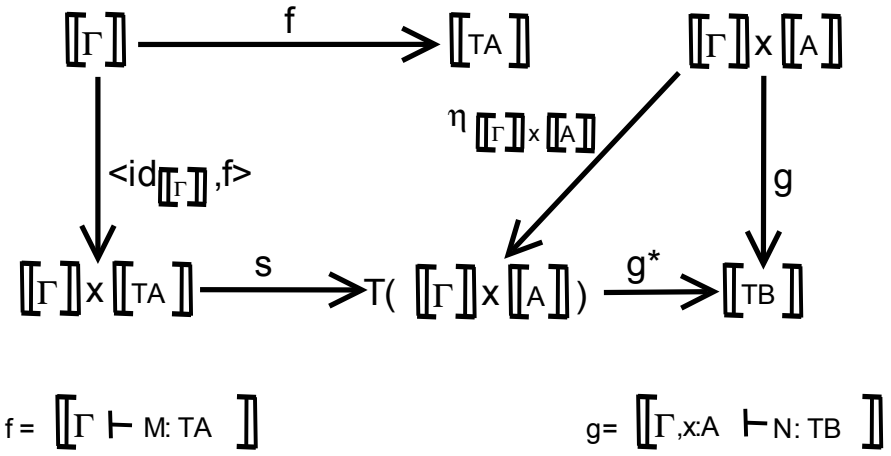


Figura 2.2: Interpretazione del let

Le equazioni per l'interpretazione dei costrutti *let* e *val* possono essere riassunte come segue:

$$[\![\Gamma \vdash val\ M : TA]\!] = [\![\Gamma \vdash M : A]\!]; \eta_{[\![A]\!]} \tag{2.2.11}$$

$$[\![\Gamma \vdash let\ x = M\ in\ N]\!] =< id_{[\![\Gamma]\!]}, [\![\Gamma \vdash M : A]\!] >; s_{[\![\Gamma]\!] \times [\![A]\!]}; [\![\Gamma, x : A \vdash N : TB]\!]^* \tag{2.2.12}$$

Nel seguito di questa tesi verranno mostrate diverse monadi e le loro applicazioni nell'informatica. L'uso del λ-calcolo computazionale, e delle equazioni definite per l'interpretazione, permetterà di definire una semantica denotazionale per diversi linguaggi di programmazione, che incorporano al loro interno diversi meccanismi computazionali.

3

Modelli computazionali

Nei precedenti capitoli sono stati introdotti i concetti basilari della teoria delle categorie, il λ-calcolo computazionale e i metodi di interpretazione di un calcolo in una categoria.

Si vogliono ora mostrare alcuni esempi di monadi semanticamente rilevanti, al fine di presentare diversi aspetti di una computazione che posso essere modellati utillizzando delle monadi appropriate. Si mostrerà inoltre come sia possibile derivare un'interpretazione per λ_C (eventualmente aumentato di una serie di costanti e primitive computazionali) all'interno di una categoria che soddisfi determinate proprietà (tra cui, in primis, l'esistenza della monade considerata all'interno di tale categoria).

Nel seguito, si userà la notazione $[\![\cdot]\!]$ per l'interpretazione dei termini nel contesto vuoto. Nei casi in cui si debba interpretare un termine in un contesto Γ si userà invece la notazione $[\![\cdot]\!]_\Gamma$; tuttavia, quando non sarà strettamente necessario, si ometterà il contesto nell'interpretazione dei termini.

3.1 Monade identità

La monade identità rappresenta l'esempio di monade più semplice esistente. Tale monade è presente in una qualsiasi categoria $\mathcal{C}$, ed è definito dal funtore identità $id\ A = A$. La trasformazione naturale $\eta_A : A \to A$ è definita come $eta_A \stackrel{\triangle}{=} id_A$. Infine, dato un morfismo $A \to B$, il morfismo f^* è dato da $f^* = f$. E' immediato verificare che le equazioni 1.6.4, 1.6.5 e 1.6.6 vengono soddisfatte per la tripla $< id,\eta,\_^* >$ così definita, e quindi che tale tripla è una tripla di Kleisli (a cui è associata, dalla proposizione 1.6.1, una monade). Le operazioni *let* e *val*, per la monade identità, vengono banalmente definite come:

$$[\![val\ M]\!] = [\![M]\!] \tag{3.1.1}$$

$$[\![let\ x = M\ in\ N]\!] = [\![M\{N/x\}]\!] \tag{3.1.2}$$

3.2 Errori

L'esempio più semplice di monade non banale che può essere esibito è dato dalla monade degli errori. Tale monade associa ad ogni oggetto A di una categoria $\mathcal{C}$ un oggetto $TA = A + 1$. Una computazione di tipo A viene intesa come la restituzione di un elemento di tipo A, oppure come l'unico elemento $* \in 1$, corrispondente ad una situazione di errore.

$$\eta_A = \iota_1 : A \to A + 1 \tag{3.2.1}$$

L'operazione $\_^*$ viene invece definita come

$$\begin{aligned} f^*(x) \quad = \quad & case\ x\ of \\ & \iota_1(a).f(a) \\ & \iota_2(*).\iota_2(*) \end{aligned} \tag{3.2.2}$$

All'interno del λ-calcolo computazionale, vengono utilizzati i costrutti let_{err} e val_{err}, la cui interpretazione è così definita (all'interno di una categoria in cui sia presente la tripla di Kleisli $(T,\eta,\_^*)$ presentata):

$$\begin{aligned} [\![val_{err}\ M]\!] \quad &= \quad \iota_1([\![M]\!]) \\ [\![let_{err}\ x = M\ in\ N]\!] \quad &= \quad case\ [\![M]\!]\ of \\ & \qquad \iota_1(a).[\![N]\!]a \\ & \qquad \iota_2(*).\iota_2(*) \end{aligned}$$

L'interpretazione data dei costrutti let_{err} e val_{err} soddisfa le equazioni 2.2.3, 2.2.4, 2.2.5, come prova la seguente proposizione:

Proposizione 3.2.1.

1. $let_{err}\ x = M\ in\ val_{err}\ x = M$
2. $let_{err}\ x = val_{err}\ M\ in\ N = N\{M/x\}$
3. $let_{err}\ y = L\ in\ (let_{err}\ x = M\ in\ N) \ = \ let_{err}\ x = (let_{err}\ y = L\ in\ M)\ in\ N\ (y \notin FV(N))$

Dimostrazione. 1. $let_{err}\ x = M\ in\ val_{err}\ x = M$.

$$\begin{aligned} [\![let_{err}\ x = M\ in\ val_{err}\ x]\!] \quad &= \quad case\ [\![M]\!]\ of \\ & \qquad \iota_1(x).(val_{err}\ x) \\ & \qquad \iota_2(*).\iota_2(*) \\ \\ &= \quad case\ [\![M]\!]\ of \\ & \qquad \iota_1(x).\iota_1(x) \\ & \qquad \iota_2(*).\iota_2(*) \\ \\ &= \quad [\![M]\!] \end{aligned}$$

2. $let_{err}\ x = val_{err}\ M\ in\ N = N\{M/x\}$.

$$\begin{aligned}
[\![let_{err}\ x = val_{err}\ M\ in\ N]\!] &= case\ val_{err}\ [\![M]\!]\ of \\
&\quad \iota_1(x).[\![N]\!]x \\
&\quad \iota_2(*).\iota_2(*) \\
&= case\ \iota_1([\![M]\!])\ of \\
&\quad \iota_1(x).[\![N]\!]x \\
&\quad \iota_2(*).\iota_2(*) \\
&= [\![N]\!]([\![M]\!]) \\
&= [\![N\{M/x\}]\!]
\end{aligned}$$

3. $let_{err}\ y = L\ in\ (let_{err}\ x = M\ in\ N)\ =\ let_{err}\ x = (let_{err}\ y = L\ in\ M)\ in\ N\ (y \notin FV(N))$.
Si semplifichino, tramite operazioni algebriche, entrambi i lati dell'equazione. Per il lato sinistro si ottiene

$$\begin{aligned}
[\![let_{err}\ y = L\ in\ let_{err}\ x = M\ in\ N]\!] &= case\ [\![L]\!]\ of \\
&\quad \iota_1(y).(let_{err}\ x = [\![M]\!]\ in [\![N]\!])y \\
&\quad \iota_2(*).\iota_2(*) \\
&= case\ [\![L]\!]\ of \\
&\quad \iota_1(y).let_{err}\ x = ([\![M]\!]y); in [\![N]\!] \\
&\quad \iota_2(*).\iota_2(*) \\
&= case\ [\![L]\!]\ of \\
&\quad \iota_1(y).case\ [\![M]\!]y\ of \\
&\quad\ \iota_1(x).[\![N]\!]x \\
&\quad\ \iota_2(*).\iota_2(*)
\end{aligned}$$

Dove $(let_{err}\ x = [\![M]\!]\ in\ [\![N]\!])y = let_{err}\ x = [\![M]\!]y\ in\ [\![N]\!])$ è vera poiché nell'equazione 2.2.5 si assume $y \notin FV(N)$.

Per il lato destro dell'equazione, invece, si ha:

$$
\begin{array}{rcl}
let_{err}\ x = (let_{err}\ y = L\ in\ M)\ in\ N & = & let_{err}\ (\\
 & & \quad case\ [\![L]\!]\ of \\
 & & \quad\ \iota_1(y).[\![M]\!]y \\
 & & \quad\ \iota_2(*).\iota_2(*)) \\
 & & \quad in [\![N]\!] \\
 & & \\
 & = & case(\\
 & & \quad case\ [\![L]\!]\ of \\
 & & \quad\ \iota_1(y).[\![M]\!]y \\
 & & \quad\ \iota_2(*).\iota_2(*)) \\
 & & \quad of \\
 & & \quad\ \iota_1(x).[\![N]\!]x \\
 & & \quad\ \iota_2(*).\iota_2(*) \\
 & & \\
 & = & case\ [\![L]\!]\ of \\
 & & \quad \iota_1(y).case\ [\![M]\!]y\ of \\
 & & \quad\ \iota_1(x).[\![N]\!]x \\
 & & \quad\ \iota_2(*).\iota_2(*) \\
 & & \quad \iota_2(*).\iota_2(*)
\end{array}
$$

I due lati dell'equazione sono equivalenti allo stesso morfismo, e quindi tra di loro equivalenti.

□

La tripla di Kleisli $< \_ + 1, \eta, \_^* >$ esibita in questa sezione può essere estesa ad una tripla $< \_ + E, \eta_E, (\_^*)_E >$, dove E non coincide più necessariamente con l'oggetto terminale 1, ma con un oggetto fissato della categoria considerata. Le trasformazioni η_E e $(\_^*)_E$ rimangono equivalenti a quelle definite per la tripla degli errori, salvo fatta eccezione per gli iniettori $\iota_1 : A \to A + E$ e $\iota_2 : E \to A + E$. In maniera analoga a quanto fatto per la proposizione 3.2.1, si dimostra che tale tripla è anch'essa di Kleisli. L'effetto computazionale associato a questa è dato dalle **eccezioni**. Anziché prevedere una sola situazione di errori, vengono messe a disposizione (tramite l'oggetto E) diverse eccezioni: **SML** fornisce un esempio tipico di linguaggio dove l'insieme delle eccezioni è molto ricco di struttura.

3.3 Lifting

Molti modelli di calcolo tipati, quali ad esempio $\lambda^{\to}_{\times}$. posseggono una proprietà di normalizzazione forte. Tale proprietà afferma che, preso un qualsiasi termine del modello, qualsiasi

strategia di riduzione porterà a ridurre il termine alla sua forma normale. In altre parole, qualsiasi programma interpretato in tale modello terminerà la sua esecuzione. E' importante notare che un modello fortemente normalizzante non possono essere rappresentate tutte le funzioni ricorsive: se così fosse, difatti, si potrebbe risolvere il problema dell'arresto verificando che il programma sia interpretabile all'interno di tale modello. Nel λ-calcolo computazionale la normalizzazione forte del sistema dipende strettamente dalla monade utilizzata per l'interpretazione dei sistemi *let* e *val*. Se si utilizza per tale interpretazione la monade identità $TA = A$, allora il sistema che si otterrà sarà equivalente al λ-calcolo semplicemente tipato, che è dimostrato essere fortemente normalizzante in [Bar92]. Differentemente, in [Cen96] viene mostrato come, utilizzando la monade delle eccezioni per l'interpretazione dei termini di λ_C, sia possibile ottenere un operatore di punto fisso Y, e quindi la possibilità di non terminazione.

Per introdurre la sola non terminazione quale effetto computazionale, si può considerare una computazione di tipo A come un valore restituito dall'insieme A, oppure come un particolare elemento $\perp$ che rappresenta la nonterminazione della computazione. Nella teoria dei domini, ad ogni insieme A viene associato un dominio piatto $A_\perp$ costituito da $A \cup \{\perp\}$, su cui viene definito l'ordinamento parziale $\sqsubseteq$ tale che $\forall a \in A_\perp, \perp \sqsubseteq a$.

Vi sono diverse categorie in cui è possibile interpretare la non terminazione (quale ad esempio **CPO**) in oggettidella forma $A_\perp$ in tali categorie è possibile definire l'endofuntore $TA = A_\perp$, che associa ad ogni oggetto A il dominio piatto $A_\perp$. Esistono inoltre una trasformazione naturale η tale che $\eta_A(a) = a \in A_\perp$ e un operatore $\_^* : \mathcal{C}[A,TB] \to \mathcal{C}[TA,TB]$ tale che $f^*(a) = f_\perp(a)$, dove $f_\perp = f[\perp \leftarrow \perp]$.

E' possibile dimostrare che la tripla $< (\_)_\perp, \eta, \_^* >$ così definita corrisponde ad una tripla di Kleisli:

Proposizione 3.3.1. *Siano $TA = A_\perp$, $\eta_A(a) = a \in A_\perp$ e $f^* = f_\perp$. Allora la tripla $< T, \eta, \_^* >$ è una tripla di Kleisli.*

Dimostrazione. E' sufficiente provare che per tale tripla, le equazioni 1.6.4, 1.6.5 e 1.6.6 sono valide. Tali prove sono semplici da ottenere:

1.6.4: Per questa equazione si ha $(\eta_A)_\perp = \eta_A[\perp \leftarrow \perp](a) = id_{A_\perp}$.

1.6.5: In questo caso si ottiene $f_\perp(\eta_A(a)) = f(a)$.

1.6.6: L'ultima equazione può essere affrontata per casi:

$$g_\perp(f_\perp(\perp)) = \perp = (g \circ f_\perp)(\perp)$$
$$g_\perp(f_\perp(a)) = (g_\perp \circ f)_\perp(a)$$

□

Dal punto di vista del λ-calcolo computazionale, vengono introdotti gli operatori $val_\perp$ e $let_\perp$.
Poiché si è dimostrato che la tripla $< (\_)_\perp, \eta_A, \_^* >$ definita in questa sezione è effettivamente una tripla di Kleisli, segue in maniera immediata che $val_\perp$ e $let_\perp$ soddisfano i tre assiomi del λ-calcolo computazionale.

3.4 Side Effects

Spesso, durante la valutazione di un programma, si è interessati anche ad altri aspetti di una computazione, oltre che al valore di ritorno. In molti casi, difatti, ad un programma viene associata la nozione di stato: uno stato può essere inteso come informazione sulla computazione del programma, relativamente a diversi aspetti di computazione. Ad esempio, uno stato può essere definito come una configurazione di memoria, così da poter osservare il comportamento di un programma non solo relativamente al valore che esso restituisce, ma anche alle modifiche che comporta sulla configurazione di memoria.

La monade dei side effects viene utilizzata per rappresentare la nozione di stato nell'interpretazione del λ-calcolo computazionale. Dato un insieme di stati S, un programma che restituisce un valore di tipo A potrà essere interpretato come un programma che, dato in input uno stato $s_0 \in S$, restituisce una coppia $< a, s_f > \in A \times S$, ovvero la coppia contenente il valore restituito dal programma (di tipo A), e lo stato finale di tipo S che si è raggiunto eseguendo tale programma partendo da uno stato iniziale s_0. Tale intuizione può essere formalizzata nella tripla di Kleisli che ha come funtore:

$$TA = (A \times S)^S \tag{3.4.1}$$

Nel metalinguaggio vengono introdotti i due costrutti $Sval$ e $Slet$, la cui interpretazione è definita come segue:

$$[\![Sval\ M]\!] = \lambda s : S. < [\![M]\!], s > \tag{3.4.2}$$

$$[\![Slet\ x = M\ in\ N]\!] = \lambda s : S. [\![N]\!](\pi_1 [\![M]\!] s)(\pi_2 [\![M]\!] s) \tag{3.4.3}$$

Per verificare che effettivamente la definizione dei costrutti $Slet$ e $Sval$ costituisce una monade, si deve mostrare che le equazioni 2.2.3, 2.2.4, 2.2.5 vengono verificate. La seguente proposizione dimostra dunque che la struttura presentata in questa sezione è effettivamente una monade:

Proposizione 3.4.1.

$Slet\ x = M\ in\ Sval\ x = M$

$Slet\ x = Sval\ M\ in\ N = N\{M/x\}$

$Slet\ y = L\ in\ (Slet\ x = M\ in\ N) = Slet\ x = (Slet\ y = L\ in\ M)\ in\ N\ (y \notin FV(N))$

La dimostrazione di questa proposizione verrà esibita in 3.5, in quanto segue come corollario di un risultato più generale, dato dalla proposizione 3.7.1.

3.5 Costruttori di monadi

L'uso del λ-calcolo computazionale permette di utilizzare le monadi per modellare diverse caratteristiche dei linguaggi di programmazione, in modo da poter definire una interpretazione dei programmi all'interno di una categoria che presenti la struttura necessaria. Tuttavia, per quanto è stato descritto finora, non è possibile combinare diversi effetti computazionali per ottenerne di nuovi. Come esempio, data la monade dei **side effects** e la monade del **lifting**, non è stato presentato nessuno strumento che permetta di combinarle in un'unica monade, che aggiunga il concetto di lifting a una computazione data solamente dai side effects. Uno degli intenti di Moggi, nell'introdurre il λ-calcolo computazionale, era quello di rendere la semantica denotazionale **modulare**. Tale possibilità è data dall'utilizzo dei **trasformatori di monadi**. Tali costruttori vengono utilizzati per trasformare una monade originaria in una nuova monade, in modo da aggiungere delle nuove caratteristiche computazionali alla prima.

Definizione 3.5.1.

Costruttore di Monadi

*Un **Costruttore di monadi**, o **Costruttore Semantico**, è una funzione $\mathcal{F}$ che, data una categoria $\mathcal{C}$ dotata di opportuna struttura, associa a ciascuna $< T,\eta,\mu >$ una nuova monade $< \mathcal{F}T,\mathcal{F}\eta,\mathcal{F}\mu >$ in $\mathcal{C}$.*

I trasformatori di monadi, introdotti da Moggi in [Mog91], permettono di definire una nuova monade (e di ottenerne automaticamente un'interpretazione dei termini all'interno di una categoria) applicando una monade originaria al trasformatore. Inoltre, dato un costruttore di monadi $\mathcal{F}$, ad esso viene associata in maniera naturale una monade data dall'applicazione del trasformatore alla monade identità, $\mathcal{F}id$. Spesso, ai costruttori semantici viene abbinata la monade del lifting presentata in sezione 3.3[1], al fine di aggiungere la possibilità di non terminazione ad effetti computazionali che non prevedono tale eventualità.
Si noti inoltre che è possibile, data una monade di base il cui funtore sia T, applicare ad essa più costruttori di monadi, così da ottenere delle strutture estremamente ricche che rappresentino effetti computazionali alquanto complessi. Dati una tripla di Kleisli $< T,\eta,\_^* >$ e due costruttori di monadi $\mathcal{F}$, $\mathcal{G}$, dalla definizione 3.5.1 si ottiene che $\mathcal{F}T$ costituisce una monade, e dunque $\mathcal{G}(\mathcal{F}T)$ sarà anch'essa una monade. Tuttavia, non è garantita la commutatività dei due costruttori semantici, i.e. è possibile trovare dei costruttori $\mathcal{F},\mathcal{G}$ per cui $\mathcal{F}(\mathcal{G}T) \neq \mathcal{G}(\mathcal{F}T)$.

[1] In realtà in tale sezione si è introdotta la tripla di Kleisli del lifting. Tuttavia, esistendo una corrispondenza uno a uno tra monadi e triple di Kleisli, è immediato ricavare da tale tripla la monade equivalente

In seguito si farà largo uso dei costruttori di monadi: difatti, sarà necessario considerare diversi effetti computazionali, combinandoli in maniera opportuna per ottenerne una rappresentazione tramite un'unica monade. In questo modo sarà possibile utilizzare il λ-calcolo computazionale per interpretare i termini che generano l'effetto computazionale considerato.

3.6 Costruttore degli errori

E' stata analizzata, nella sezione 3.2, la monade degli errori. Spesso, tuttavia, si è interessati a definire il concetto di errore su una computazione già definita. Tale possibilità viene data dal costruttore di monadi

$$(\mathcal{E}T)A = T(A + 1) \tag{3.6.1}$$

Se si applica, ad esempio, a tale costruttore la monade del lifting $TA = A_\perp$, si otterrà come risultato la monade $TA = (A + 1)_\perp$, ovvero una computazione di tipo A è data dalla possibilità di non terminazione, oppure da un valore che può essere di tipo A oppure un errore.

Data una monade T, con le relative operazioni val_T, e let_T, si definiscono gli operatori $val_{\mathcal{E}}T$) e $let_{\mathcal{E}T}$ come

$$[\![val_{\mathcal{E}T}\ M]\!] \quad = \quad val_T\ \iota_1\ [\![M]\!] \tag{3.6.2}$$

$$\begin{aligned} [\![let_{\mathcal{E}T}\ x = M\ in\ N]\!] \quad = \quad & let_T\ x' = [\![M]\!]\ in \\ & case\ x'\ of \\ & \iota_1(x).[\![N]\!]x \\ & \iota_2(*).val_T\ \iota_2(*) \end{aligned} \tag{3.6.3}$$

L'operatore $val_{\mathcal{F}T}$ stabilisce che il lifting di un termine di tipo A è dato dal lifting, rispetto alla monade, T dell'interpretazione di tale termine in $A + 1$. L'operatore $let_{\mathcal{F}T}$, invece, mostra che la composizione di termini può essere ottenuta dalla composizione definita per la monade T, effettuando una case analysis per distinguere i valori veri e propri dagli errori.

Proposizione 3.6.1. *Il costruttore delle definizioni 3.6.1, 3.6.2, 3.6.3 è un costruttore di monadi*

Dimostrazione. Si supponga di avere una monade T, con relativi operatori let_T e val_T. Essendo T una monade, essa soddisfa gli assiomi di λ_C. Per mostrare che $\mathcal{E}T$ forma una monade rispetto agli operatori $val_{\mathcal{E}T}$ e $let_{\mathcal{E}T}$, è sufficiente dimostrare che anche essa soddisfa tali assiomi. Per convenienza di notazione, si indicheranno $val_{\mathcal{E}T}$ e $let_{\mathcal{E}T}$ con $Eval$ ed $Elet$.

2.2.3: $Elet\ x = M\ in\ Eval\ x = M$

$$\begin{aligned}
[\![Elet\ x = M\ in\ Eval\ x]\!] &= let_T\ x' = [\![M]\!]\ in \\
&\quad case\ x'\ of \\
&\quad\ \iota_1(x).val_T\ \iota_1(x) \\
&\quad\ \iota_2(*).val_T\ \iota_2(*) \\
&= let_T\ x' = [\![M]\!]\ in\ val_T \\
&\quad case\ x'\ of \\
&\quad\ \iota_1(x).\iota_1(x) \\
&\quad\ \iota_2(*).\iota_2(*) \\
&= let_T x' = [\![M]\!]\ in\ val_T\ x' \\
&= [\![M]\!]
\end{aligned}$$

2.2.4: $Elet\ x = Eval\ M\ in\ N\ =\ N\{M/x\}$

$$\begin{aligned}
[\![Elet\ x = Eval\ M\ in\ N]\!] &= let_T\ x' =\ val_T\ \iota_1\ [\![M]\!]\ in \\
&\quad case\ x'\ of \\
&\quad\ \iota_1(x).[\![N]\!]x \\
&\quad\ \iota_2(*).val_T\ \iota_2(*) \\
&= case\ \iota_1([\![M]\!])\ in \\
&\quad\ \iota_1(x).[\![N]\!]x \\
&\quad\ \iota_2(*).val_T\ \iota_2(*) \\
&= [\![N]\!]([\![M]\!]) \\
&= [\![N\{M/x\}]\!]
\end{aligned}$$

2.2.5: $Elet\ y = L\ in\ Elet\ x = M\ in\ N\ =\ Elet\ x = (Elet\ y = L\ in\ M)\ in\ N$, con $y \notin FV(N)$.

$$\begin{aligned}
[\![Elet\ x = (Elet\ y = L\ in\ N)]\!] &= let_T\ x' = (let_T\ y' = [\![L]\!]\ in \\
&\quad case\ y'\ of \\
&\quad\ \iota_1(y).[\![M]\!]y \\
&\quad\ \iota_2(*).val_T\ \iota_2(*)) \\
&\quad in\ case\ x'\ of \\
&\quad\ \iota_1(x).[\![N]\!]x \\
&\quad\ \iota_2(*).val_T\ \iota_2(*)
\end{aligned}$$

Posti

$$\begin{aligned} M'(y') &= case\ y'\ of \\ &\quad \iota_1(y).[\![M]\!]y \\ &\quad \iota_2(*).val_T\ \iota_2(*) \end{aligned}$$

$$\begin{aligned} N'(x') &= case\ x'\ of \\ &\quad \iota_1(x).[\![N]\!]x \\ &\quad \iota_2(*).val_T\ \iota_2(*) \end{aligned}$$

Si riscrive tale termine come

$$let_T\ x' = (let_T\ y' = L\ in M')\ in N'$$

Pertanto

$$
\begin{aligned}
let_T\ x' = (let_T\ y' = [\![L]\!]\ in M')\ in N' &= let_T\ y' = [\![L]\!]\ in\ let_T\ x' = M'\ in\ N' \\
&= let_T\ y' = [\![L]\!]\ in \\
&\quad let_T\ x' = (\\
&\quad\ case\ y'\ of \\
&\quad\quad \iota_1(y).[\![M]\!]y \\
&\quad\quad \iota_2(*).val_T\ \iota_2(*)) \\
&\quad in\ case\ x'\ of \\
&\quad\ \iota_1(x).[\![N]\!]x \\
&\quad\ val_T\ \iota_2(*) \\
&= let_T\ y' = [\![L]\!]\ in \\
&\quad case\ y'\ of \\
&\quad\ \iota_1(y).let_T\ x' = [\![M]\!]y\ in\ case\ x'\ of \\
&\quad\quad \iota_1(x).[\![N]\!]x \\
&\quad\quad \iota_2(x).val_T\ \iota_2(*) \\
&\quad\ \iota_2(*).let_T\ x' = val_T\ \iota_2(*)\ in\ case\ x'\ of \\
&\quad\quad \iota_1(x).[\![N]\!]x \\
&\quad\quad \iota_2(*).val_T\ \iota_2(*) \\
&= let_T\ y' = [\![L]\!]\ in \\
&\quad case\ y'\ of \\
&\quad\ \iota_1(y).let_T\ x' = [\![M]\!]y\ in\ case\ x'\ of \\
&\quad\quad \iota_1(x).[\![N]\!]x \\
&\quad\quad \iota_2(x).val_T\ \iota_2(*) \\
&\quad\ \iota_2(*).val_T\ \iota_2(*) \\
&= [\![Elet\ y = L\ in\ Elet\ x = M\ in\ N]\!]
\end{aligned}
$$

□

3.7 Costruttore dei side effects

Nella sezione 3.4 si è introdotta la monade dei side effects. Vediamo ora che tale monade può essere ottenuta da un trasformatore di monadi, detto appunto dei side effects, dato

da:

$$(\mathcal{S}T)A = T(A \times S)^S \tag{3.7.1}$$

I costrutti $let_{\mathcal{S}T}$ e $val_{\mathcal{S}T}$ della monade ottenuta applicando una monade T al costruttore $\mathcal{S}$ vengono così definiti:

$$[\![val_{\mathcal{S}T}\ M]\!] = \lambda s : S.val_T < [\![M]\!],s > \tag{3.7.2}$$
$$[\![let_{\mathcal{S}T}\ x = M\ in\ N]\!] = \lambda s : S.let_T\ < x,s' >= [\![M]\!]s\ in\ [\![N]\!]s' \tag{3.7.3}$$

Applicando a tale trasformatore la monade identità, si ottiene la monade dei side effects $T'A = (A \times S)^S$. Il trasformatore 3.7.1 permette di aggiungere degli effetti computazionali a dei programmi che già posseggono una nozione di stato al loro interno. Ad esempio, è possibile applicare il trasformatore 3.7.1 alla monade del lifting per ottenere una monade che rappresenta l'effetto computazionale dato da una funzione che restituisce, dato uno stato in input, una coppia contenente il valore restituito da un programma e lo stato in cui tale programma termina, oppure $\perp$. Tale monade, in particolare, è utile nel caso in cui si voglia dare un'interpretazione dei termini di un linguaggio che preveda la non terminazione dei programmi come funzioni che mappano una configurazione di memoria iniziale in una configurazione di memoria finale.

Per dimostrare che la funzione $\mathcal{S}$ è effettivamente un trasformatore di monadi, è sufficiente verificare che, data una monade T, $\mathcal{S}T$ è ancora una monade. Ciò è espresso dalla seguente proposizione:

Proposizione 3.7.1. *Sia data una monade T, i.e. let_T e val_T soddisfano le equazioni 2.2.3, 2.2.4, 2.2.5: allora tali equazioni vengono soddisfatte anche da $\mathcal{S}T = T(A \times S)^S$.*

Dimostrazione. Per semplicità di notazione, si porrà $T' = \mathcal{S}\mathcal{T}$.

1.

$$\begin{aligned}
[\![let_{T'}\ x = M\ in\ val_{T'}\ x]\!] &= \lambda s : S.let_T\ < x,s' >= [\![M]\!]s\ in\ (\lambda s' : S.val_T\ < [\![x]\!]_{x:A},s' >)s' \\
&=_\beta \lambda s : S.let_T\ < x,s' >= [\![M]\!]s\ in\ val_T\ < [\![x]\!]_{x:A},s' > \\
&= \lambda s : S.[\![M]\!]s \\
&=_\eta [\![M]\!]
\end{aligned}$$

2.

$$\begin{aligned}
[\![let_{T'}\ x = val_{T'}\ M\ in\ N]\!] &= \lambda s : S.let_T\ < x,s' >= (\lambda s'' : S.val_T < [\![M]\!],s'' >)s\ in [\![N]\!]s' \\
&=_\beta \lambda s : S.let_T\ < x,s' >= val_T\ < [\![M]\!],s >\ in\ [\![N]\!]s' \\
&= \lambda s : S.[\![N\{M/x\}]\!]s \\
&=_\eta [\![N\{M/x\}]\!]
\end{aligned}$$

3.

$$
\begin{array}{rl}
 & let_{T'}\ y = [\![L]\!]\ in\ (let_{T'}\ x = [\![M]\!]\ in\ [\![N]\!]) \\
= & \lambda s : S.let_T\ < y,s' >= [\![L]\!]s\ in\ (\lambda s : S.let_T\ < x,s'' >= [\![M]\!]s\ in [\![N]\!]s'') \\
=_\beta & \lambda s : S.let_T\ < y,s' >= [\![L]\!]s\ in\ (let_T\ < x,s'' >= [\![M]\!]s'\ in [\![N]\!]s'') \\
= & \lambda s : S.let_T\ < x,s'' >= (let_T\ < y,s' >= [\![L]\!]s\ in\ [\![M]\!]s')\ in\ [\![N]\!]s'' \\
=_\beta & \lambda s : S.let_T\ < x,s'' >= ((\lambda s : S.let_T\ < y,s' >= [\![L]\!]s)s\ in\ [\![M]\!]s')\ in\ [\![N]\!]s'' \\
= & [\![let_{\mathcal{FT}}\ x = (let_{\mathcal{FT}}\ y = L\ in\ M)\ in\ N]\!]
\end{array}
$$

□

Corollario 3.7.2. *Il funtore $TA = (A \times S)^S$, insieme ai costrutti Sval e Slet, costituisce una monade*

Dimostrazione. E' già stato osservato che $\mathcal{S}id$ corrisponde alla monade dei side effects. La proposizione 3.7.1 stabilisce dunque che essa è effettivamente una monade. □

In seguito, le operazioni *let* e *val* per il costruttore di monadi $\mathcal{S}T$ definito verranno denotate con $Slet$ e $Sval$.

3.8 Context readers

L'ultimo esempio che si vuole presentare è costituito dai context readers. Scelta una monade T che rappresenti un effetto computazionale, è possibile che la computazione di un termine $M : A$ dipenda da un contesto nel quale essa viene eseguita. Se si rappresenta con Q l'insieme dei possibili contesti, è possibile dunque rappresentare un effetto computazionale sensibile al contesto come una funzione che, preso in input un contesto, restituisce una computazione sul tipo A. Si definisce dunque il costruttore dei **context readers** come

$$(\mathcal{Q}T)A = (TA)^Q \tag{3.8.1}$$

I costrutti $let_{\mathcal{Q}T}$ e $val_{\mathcal{Q}T}$ per tale costruttore sono semplici da ottenere. Se si considera un termine $M : A$, il lifting di tale termine corrisponderà a prendere in input un contesto, procedendo con il lifting del termine (rispetto alla monade T utilizzata come argomento del costruttore): ovvero, il contesto preso in input viene ignorato. La composizione di due termini $let_{\mathcal{Q}T}\ x = M\ in\ N$ può essere invece ottenuta prendendo in input un contesto, e quindi effettuando la composizione (rispetto alla monade T) delle computazioni ottenute passando il contesto preso in ingresso ai singoli termini M ed N. Matematicamente, tali operazioni vengono descritte come

$$val_{\mathcal{Q}T}\ M = \lambda q : Q.val_T\ M \tag{3.8.2}$$

$$let_{\mathcal{Q}T}\ x = M\ in\ N = \lambda q : Q.let_T\ x = Mq\ in\ Nq \tag{3.8.3}$$

Per semplicità di notazione, si indicheranno gli operatori $let_{\mathcal{Q}T}$ e $val_{\mathcal{Q}T}$ con $QRlet$ e $QRval$

Proposizione 3.8.1. *Il costruttore $\mathcal{Q}$, equipaggiato con gli operatori $QRlet$ e $QRval$, è un costruttore di monadi.*

Dimostrazione. **2.2.3:** $QRlet\ x = M\ in\ QRval\ x\ =\ M$

$$\begin{array}{rcl} [\![QRlet\ x = M\ in\ val\ x]\!] & = & \lambda q.let_T\ x = [\![M]\!]q\ in\ (\lambda q.val_T\ x)q \\ & =_\beta & \lambda q.let_T\ x = [\![M]\!]q\ in\ val_T\ x \\ & = & \lambda q.[\![M]\!]q \\ & =_\eta & [\![M]\!] \end{array}$$

2.2.4: $QRlet\ x = QRVal\ M\ in\ N\ =\ N\{M/x\}$

$$\begin{array}{rcl} [\![Rlet\ x = QRVal\ M\ in\ N]\!] & = & \lambda q.let_T\ x = (\lambda q.val_T\ [\![M]\!])q\ in\ [\![N]\!]q \\ & =_\beta & \lambda q.let_T\ x = val_T\ [\![M]\!]\ in\ [\![N]\!]q \\ & = & \lambda q.[\![N\{M/x\}]\!]q \\ & =_\eta & [\![N\{M/x\}] \end{array}$$

2.2.5: $QRlet\ y = L\ in\ QRlet\ x = M\ in\ N\ =\ QRlet\ x = (QRlet\ y = L\ in\ M)\ in\ N$, con $y \notin FV(N)$.

$$\begin{array}{rcl} [\![QRlet\ y = L\ in\ QRlet\ x = M\ in\ N]\!] & = & \lambda q.let_T\ y = [\![L]\!]q\ in(\lambda q.let_T\ x = [\![M]\!]q\ in\ [\![N]\!]q)q \\ & =_\beta & \lambda q.let_T\ y = [\![L]\!]q\ inlet_T\ x = [\![M]\!]q\ in\ [\![N]\!]q \\ & = & \lambda q.let_T\ x = (let_T\ y = [\![L]\!]q\ in[\![M]\!]q)\ in\ [\![N]\!]q \\ & =_\beta & \lambda q.let_T\ x = (\lambda q.let_T\ y = [\![L]\!]q\ in[\![M]\!]q)q\ in[\![N]\!]q \\ & = & [\![QRlet\ x = (QRlet\ y = L\ in\ M)\ in\ N]\!] \end{array}$$

□

Se si applica al costruttore semantico $\mathcal{Q}$ la monade identità, si ottiene una monade $QA = A^Q$, i cui operatori let_Q e val_Q sono così definiti:

$$\begin{array}{rcl} val_Q\ M & = & \lambda q : Q.M \\ let\ x = M\ in\ N & = & \lambda q : Q.N(Mq)q \end{array}$$

Come corollario alla proposizione si ottiene il seguente:

Corollario 3.8.2. *Il funtore $QA = A^Q$, equipaggiato con le operazioni let_Q e val_Q, costituisce una monade.*

3.9 Non commutatività dei costruttori semantici

Precedentemente, si è affermato che, in generale, l'applicazione di più costruttori di monadi non costituisce un'operazione commutativa. Per mostrare la validità di tale affermazione, è sufficiente esibire due costruttori $\mathcal{F}$,$\mathcal{G}$ per i quali $\mathcal{F}(\mathcal{G}T) \neq \mathcal{G}(\mathcal{F}T)$.

Si scelgano rispettivamente $\mathcal{F}$ e $\mathcal{G}$ come i costruttori dei side effects e degli errori, i.e. $\mathcal{F} = \mathcal{S}$ e $\mathcal{G} = \mathcal{E}$. Data una generica monade T, si ottiene:

$$\begin{aligned}(\mathcal{E}(\mathcal{S}T))A &= (\mathcal{S}T)(A+1) &= T((A+1)\times S)^S \\ (\mathcal{S}(\mathcal{E}T))A &= (\mathcal{E}T)(A\times S)^S &= T((A\times S)+1)^S\end{aligned}$$

Nel primo caso, l'effetto computazionale di un tipo A può essere descritto come: **Dato uno stato di input $s \in S$, restituisci una computazione (definita da T) su una coppia il cui primo elemento sia un valore di tipo A o un errore, e il secondo elemento sia uno stato di output**.
Nel secondo caso, invece, l'effetto computazionale ottenuto sui valori di tipo A è dato da: **Dato uno stato di input $s \in S$, restituisci una computazione su un errore, oppure su una coppia contenente un valore di tipo A e uno stato di output**. Tale esempio mostra che l'ordine in cui i costruttori semantici vengono applicati ad una monade nfluenza la natura dell'effetto computazionale modellato: sebbene in entrambi i casi nella computazione vengano definiti un concetto di errore e un concetto di stato, questi vengono utilizzati per ricoprire un ruolo differente all'interno delle due monadi esibite.

4

Interpretazione di un linguaggio imperativo

L'interpretazione dei costrutti *let* e *val* all'interno di una categoria $\mathcal{C}$ contenente una monade T permette di definire una semantica computazionale per i termini del metalinguaggio λ_C. Diversamente da quanto si effettua classicamente in semantica, interpretando i programmi come funzioni che restituiscono un valore, l'interpretazione dei termini del λ-calcolo computazionale sarà costituito da **computazioni**, dove la monade T utilizzata per interpretare i termini descrive la forma degli effetti computazionali considerati.

In questo capitolo si mostrerà una semantica categoriale per un linguaggio di programmazione imperativo; ad ogni programma viene associata una computazione sugli stati della memoria. In questo caso si è interessati ai cambiamenti delle configurazioni di memoria provocati dall'esecuzione di un programma: poiché il linguaggio di programmazione che verrà usato come riferimento prevede la possibilità di ottenere dei programmi non terminanti, la monade che verrà utilizzata per l'interpretazione è data da $TA = (A \times S)^S_\perp$, dove S rappresenta il dominio degli stati di memoria.

La definizione della semantica denotazionale per un dato linguaggio di programmazione viene divisa in differenti passi:

- Si assume di avere un insieme di primitive computazionali, la cui segnatura viene qui denotata con Σ. Il metalinguaggio dato dal λ-calcolo computazionale, aumentato delle operazioni definite in Σ, viene denotato con $\lambda_C(\Sigma)$.
- Si definisce una traduzione $(\!| \cdot |\!) : C \to \lambda_C(\Sigma)$ dai programmi del linguaggio di programmazione al metalinguaggio.
- Data una categoria oggetto $\mathcal{C}$ contenente una monade T (utilizzata per l'interpretazione degli operatori *let* e *val* del metalinguaggio), si definisce l'interpretazione $[\![\cdot]\!]$ delle primitive computazionali contenute in Σ. Spesso è necessario porre diversi vincoli sulla struttura della categoria $\mathcal{C}$, e mostrare che esiste una categoria concreta (diversa dalla categoria $\mathbf{1}$, contenente un solo oggetto e un solo morfismo) all'interno del quale viene effettuata l'interpretazione dei termini del metalinguaggio.

- Dato un programma C del linguaggio di programmazione originario, la sua semantica sarà data da $[\![(\!|C|\!)]\!]$.

Il processo che permette di definire la semantica di un dato linguaggio di programmazione è dunque composto di 3 diversi livelli:

1. Il linguaggio di programmazione originario.
2. Il metalinguaggio $\lambda_C(\Sigma)$.
3. La categoria

Per mostrare nel dettaglio come il processo di definizione della semantica avvenga, verrà esibito un esempio in cui il linguaggio di programmazione preso in considerazione sia **imperativo** e **Turing Completo**.

4.1 Il linguaggio: TinyC

Il linguaggio che si prenderà in considerazione è una versione semplificata del ben noto linguaggio di programmazione **C**. Si assumerà di avere un solo tipo di variabili, corrispondenti ai numeri naturali, e le costanti $0,1,\cdots,n,\cdots$. Un insieme di operatori binari θ verrà utilizzato per la definizione delle espressioni. Le operazioni basilari che verranno permesse sono data dall'assegnamento $x := E$, dove x è una variabile e E è un'espressione, da una primitiva *skip* che non modifica lo stato di memoria del programma, oltre che dagli operatori di controllo del flusso di esecuzione e di iterazione, if e $while$. Infine, essendo tale linguaggio imperativo, è prevista la possibilità di concatenare più comandi sequenzialmente. La **BNF** che descrive il linguaggio descritto, denominato **TinyC**, è la seguente:

$$\begin{array}{lll} \mathrm{E} & ::= & \mathrm{n} \mid \mathrm{x} \mid \mathrm{E}_1\ \theta\ \mathrm{E}_2 \\ \mathrm{C} & ::= & \mathrm{skip} \mid \mathrm{x} := \mathrm{E} \mid \mathrm{C}_1; \mathrm{C}_2 \mid \mathrm{if\ E\ then\ C}_1\ \mathrm{else\ C}_2 \mid \mathrm{while\ E\ do\ C} \end{array}$$

4.2 Il metalinguaggio

Il metalinguaggio che verrà utilizzato è dato dal λ-calcolo computazionale, opportunamente aumentato con una serie di operazioni che rappresentano le primitive computazionali utilizzate per modellare i diversi effetti presenti in **TinyC**.

Si assumerà di disporre, per ogni costante n di **TinyC**, di una costante computazionale n nel metalinguaggio: ognuna di queste costanti avrà tipo nat. Allo stesso modo, per ogni variabile x si assumerà di avere una variabile x nel metalinguaggio. Inoltre, per ogni operatore $\theta \in \Theta$ di **TinyC**, si assumerà di avere un operatore $\underline{\theta} : nat \times nat \to nat$ nel metalinguaggio. Ciò renderà semplice la traduzione delle espressioni da **TinyC** nel metalinguaggio.

Il metalinguaggio include i seguenti costrutti:

- Lettura del valore di una variabile: per ogni variabile x, si introduce nel metalinguaggio la primitiva $read_x : Tnat$. Tale operazione, intuitivamente, restituisce una computazione su di un numero intero: nel caso in cui l'interpretazione avvenga utilizzando la monade $TA = (A \times S)^S_\perp$, l'operazione $read_x$ restituisce una funzione che, dato uno stato $s \in S$, restituisce una coppia $< n,s' >$, dove $n : nat$,$s' : S$. L'effetto computazionale associato ai termini verrà definito formalmente al momento di definire l'interpretazione dei termini del metalinguaggio all'interno di una categoria.

- Memorizzazione di un valore in una variabile: per ogni variabile x, si introduce nel metalinguaggio la primitiva $upd_x : nat \to T1$. Intuitivamente, una computazione sul tipo 1 corrisponde a una funzione che non restituisce valore. Quando verrà definita la traduzione dei programmi di **TinyC** nel metalinguaggio, si osserverà che tutti i comandi verranno tradotti in un termine avente tipo $T1$. Questa scelta corrisponde all'identificare la traduzione dei programmi di **TinyC** come dei termini che producono un effetto computazionale, ma non restituiscono alcun valore.

- Un'operazione di controllo del flusso di esecuzione: per ogni tipo A del metalinguaggio, viene introdotto un costrutto $cond_A : nat \times A \times A \to A$.

- Un operatore di astrazione: $fn\ x : A \Rightarrow M$. Questo operatore viene utilizzato per introdurre nel linguaggio la possibilità di definire delle funzioni. Se $M : B$, allora $fn\ x : A \Rightarrow M : A \to B$.

- L'applicazione di funzioni: se $M : A \to B$ ed $N : A$, allora sarà possibile applicare N ad M, ottenendo così il termine $MN : B$.

- Dato un termine di tipo $T1$, un operatore di punto fisso $fix : (T1 \to T1) \to T1$. L'uso combinato delle operazioni $fn\ x : A \Rightarrow M$ e fix_A verrà utilizzato per la traduzione del comando *while* di **TinyC**.

Inoltre, il metalinguaggio conterrà i costrutti *let* e *val* tipici del λ-calcolo computazionale. La definizione formale del metalinguaggio è data dalle seguenti regole di tipaggio:

$$\frac{}{\Gamma \vdash * : 1} \qquad \frac{}{\Gamma \vdash n : nat}$$

$$\frac{}{\Gamma \vdash read_x : Tnat} \qquad \frac{\Gamma \vdash m : nat \quad \Gamma \vdash n : nat}{\Gamma \vdash \underline{\theta}(m,n) : nat}$$

$$\frac{\Gamma \vdash n : nat}{\Gamma \vdash upd_x\ n : T1} \qquad \frac{\Gamma \vdash n : nat \quad \Gamma \vdash M : A \quad \Gamma \vdash N : A}{\Gamma \vdash cond_A(n,M,N) : A}$$

$$\frac{\Gamma,x : A \vdash M : B}{\Gamma \vdash fn\ x : A \Rightarrow M : A \to B} \qquad \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$$

$$\frac{\Gamma \vdash M : T1 \to T1}{\Gamma \vdash fix(M) : T1} \qquad \frac{\Gamma \vdash M : A}{\Gamma \vdash val\ M : TA}$$

$$\frac{\Gamma \vdash M : TA \quad \Gamma,x : A \vdash N : TB}{\Gamma \vdash let\ x = M\ in\ N : TB}$$

Viene inoltre effettuata l'assunzione che le segueti uguaglianze vengano soddisfatte per i termini del metalinguaggio:

$$cond_A(0,M,N) = M \tag{4.2.1}$$
$$cond_A(n,M,N) = N \ \ (n \neq 0) \tag{4.2.2}$$
$$(fn\ x : A \Rightarrow M)N = M\{N/x\}\ (N : A) \tag{4.2.3}$$
$$M\ (fix\ M) = fix\ M\ (M : A \to A) \tag{4.2.4}$$

L'interpretazione dei termini all'interno di una categoria dovrà dunque soddisfare i requisiti imposti da tali equazioni, oltre che dalle equazioni che definiscono let e val come elementi costitutivi di una monade.

4.3 Dal linguaggio al metalinguaggio

Definiti formalmente il linguaggio di cui si vuole specificare un'interpretazione, e il metalinguaggio che verrà usato per l'interpretazione all'interno di una categoria $\mathcal{C}$, è necessario esplicitare una traduzione $(\![\cdot]\!)$ da **TinyC** al metalinguaggio. Verrà prima discussa la traduzione delle espressioni E, per poi definire la traduzione dei comandi C.

Una espressione di **TinyC** restituisce un valore intero. Nel metalinguaggio, la valutazione di un'espressione corrisponde dunque ad una computazione sul tipo degli interi *nat*. Formalmente, la traduzione di un'espressione E corrisponde a un termine del metalinguaggio $(\!|\mathrm{E}|\!) : T nat$.
Una costante n di **TinyC** verrà tradotta nella rispettiva costante n del metalinguaggio. Tuttavia, tali costanti hanno tipo *nat*, contrariamente a quanto appena richiesto (ovvero, essendo n un'espressione, che $(\!|\mathrm{n}|\!) : T nat$). Tale problema viene risolto ponendo

$$(\!|\mathrm{n}|\!) = val\ n : T\ nat \tag{4.3.1}$$

La presenza di una variabile all'interno di un'espressione corrisponde alla lettura della configurazione di memoria in tale variabile. Nella sezione 4.2 è stata introdotta appositamente la famiglia di primitive computazionali $read_x$, di tipo $T nat$. La traduzione dell'espressione x corrisponde a

$$(\!|\mathrm{x}|\!) = read_x : T\ nat \tag{4.3.2}$$

Infine, un'espressione può essere data da $\mathrm{E}_1\ \theta\ \mathrm{E}_2$. La traduzione di tale espressione sarà definita a partire dalle traduzioni $(\!|\mathrm{E}_1|\!)$ e $(\!|\mathrm{E}_2|\!)$. La primitiva $\underline{\theta}$ del metalinguaggio prende in input una coppia di interi, e restituisce un intero. A tale risultato dovrà essere applicato il costrutto *val* secondo il vincolo espresso per cui $(\!|\mathrm{E}|\!) : T nat$. Si ottiene

$$(\!|\mathrm{E}_1\ \theta\ \mathrm{E}_2|\!) = let\ m = (\!|\mathrm{E}_1|\!)\ in\ let\ n = (\!|\mathrm{E}_2|\!)\ in\ val\ \underline{\theta}(m,n) : T\ nat \tag{4.3.3}$$

Infine, si dovrà definire la traduzione di un comando C di **TinyC**. Tali comandi non restituiscono alcun valore, ma corrispondono comunque a delle computazioni. Si richiederà, dunque, che la traduzione di un comando corrisponda a un termine di tipo $T1$.
Il comando skip corrisponde ad un comando in cui l'effetto computazionale è nullo: in questo caso, si pone banalmente

$$(\!|\mathrm{skip}|\!) = val * : T\ 1 \tag{4.3.4}$$

L'assegnamento x :=E, invece, corrisponde all'aggiornamento della locazione di memoria x con il valore restituito dalla valutazione dell'espressione E. La traduzione di tale comando sfrutterà la primitiva computazionale upd_x: si ottiene:

$$(\!|\mathrm{x} := \mathrm{E}|\!) = let\ n = (\!|\mathrm{E}|\!)\ in\ upd_x\ n : T\ 1 \tag{4.3.5}$$

La concatenazione di programmi C_1 ; C_2 è data dall'applicazione del costrutto *let*, in quanto si procederà a valutare la traduzione del comando $(\!|\mathrm{C}_1|\!)$, e poi a valutare la traduzione del comando $(\!|\mathrm{C}_2|\!)$.

$$(\!|\mathrm{C}_1\ ;\ \mathrm{C}_2|\!) = let\ x = (\!|\mathrm{C}_1|\!)\ in\ (\!|\mathrm{C}_2|\!) : T\ 1 \tag{4.3.6}$$

Il flusso di esecuzione può essere controllato nel metalinguaggio facendo uso della famiglia di costrutti $cond_A$. In particolare, la traduzione di un costrutto if E then C_1 else C_2 può essere effettuata utilizzando la primitiva $cond_{T1}$. Difatti, il comando if E then C_1 else C_2

corrisponde al valutare l'espressione E, e quindi a selezionare l'opportuno comando da eseguire.

$$(\!|\text{if E then } C_1 \text{ else } C_2|\!) = let\ n = (\!|\text{E}|\!)\ in\ cond_{T1}(n,(\!|C_1|\!),(\!|C_2|\!)) : T\ 1 \tag{4.3.7}$$

La traduzione di un comando while E do C è certamente la più difficile. É noto che tale comando soddisfa la seguente uguaglianza:

$$\text{while E do C} = \text{if E then (C; while E do C) else skip}$$

Tale relazione dovrà essere soddisfatta anche dalla traduzione del costrutto while: si ottiene

$$(\!|\text{while E do C}|\!) = let\ n = (\!|\text{E}|\!)\ in\ cond_{T1}(n,let\ x = (\!|\text{C}|\!)\ in\ (\!|\text{while E do C}|\!), val\ *)$$

Tale equazione è definita in maniera ricorsiva. É possibile esplicitarne il risultato facendo uso dell'operatore di punto fisso fix. Il risultato che si ottiene viene mostrato qui di seguito:

$$(\!|\text{while E do C}|\!) = fix(fn\ M : T1 \Rightarrow let\ n = (\!|\text{E}|\!)\ in\ cond_{T1}(n,let\ x = (\!|\text{C}|\!)\ in\ M, val\ *)) : T1 \tag{4.3.8}$$

Infine, la traduzione fornita gode della seguente proprietà:

Teorema 4.3.1 (Adeguatezza Statica)**.** *Scelta arbitrariamente un'espressione E di* ***TinyC****, si ha* $(\!|E|\!) : Tnat$*. Scelto arbitrariamente un comando C di* ***TinyC****, si ha* $(\!|C|\!) : T1$*.*

4.4 La categoria

L'ultimo livello che si deve introdurre, per ottenere una semantica di **TinyC**, è dato dalla categoria in cui i termini del metalinguaggio verranno interpretati. E' necessario effettuare un'analisi della struttura che tale categoria dovrà presentare, in quanto l'interpretazione che mapperà i termini del metalinguaggio nella categoria dovrà soddisfare la proprietà di correttezza, vale a dire

$$M = N \Rightarrow [\![M]\!] = [\![N]\!]$$

Inoltre, si dovrà esibire un esempio di categoria concreta che presenti la struttura richiesta. Tale categoria dovrà essere differente dalla categoria **1**. In tal caso, difatti, si otterrebbe banalmente

$$\forall M,N \in \lambda_c(\Sigma).[\![M]\!] = [\![N]\!]$$

Ottenendo così una teoria inconsistente.
Verrà dunque esaminata la struttura richiesta ad una categoria $\mathcal{C}$ affinché sia possibile esibire una interpretazione dei termini del metalinguaggio in $\mathcal{C}$.

L'interpretazione dovrà essere data in una categoria contenente la monade $TA = (A \times S)^S_\perp$. Tale categoria dovrà inoltre essere provvista di un oggetto per interpretare i termini di tipo *nat* del metalinguaggio: tale oggetto dovrà essere un **natural number object** (**nno**)

ed un oggetto terminale. Si noti che un **nno** N è soluzione dell'equazione $X \simeq X + 1$, ovvero l'oggetto: N può dunque essere scritto nella forma

$$\mu X.X + 1$$

dove $\mu X.f(X)$ corrisponde al minimo oggetto X isomorfo ad $f(X)$.
Ciò permette di effettuare l'analisi per casi sui **nno**: Per quanto detto esiste un isomorfismo $\alpha = [succ,z] : N + 1 \to N$. L'elemento globale $z : 1 \to N$ corrisponde alla definizione del naturale *zero*, mentre $succ : N \to N$ corrisponde alla funzione successore. Dati $n \in N$, $f : N \to A$, $g : 1 \to A$, è possibile costruire il morfismo

$$\begin{array}{c} case\ n\ of \\ z.g(*) \\ succ(n).f(n) \end{array} \tag{4.4.1}$$

Tale morfismo corrisponde a verificare se un naturale n è pari a 0, nel qual caso viene applicato il morfismo g: in caso contrario, verrà applicato il morfismo f ad n.
Si assume di avere, all'interno della categoria, un oggetto V che contenga le variabili di **TinyC**. Le variabili verranno dunque interpretate come elementi dell'oggetto V. Inoltre, nella monade $TA = (A \times S)^S_\perp$, gli stati corrispondono alle configurazioni di memoria, ovvero a funzioni dall'oggetto delle variabili V nel natural number object. Si scelga dunque N come *nno*, e si ponga $S = \mathbb{N}^{\mathcal{V}}$.
Poiché nel metalinguaggio si è introdotto un operatore di astrazione $fn\ x : A \to B$, si deve richiedere che la categoria $\mathcal{C}$ sia cartesiana chiusa.
Infine, per l'operatore fix del metalinguaggio, si deve assumere che ogni endomorfismo $T1 \to T1$ dell'oggetto $T1$ ammetta un punto fisso. Essendo $T1 = (1 \times S)^S_\perp$ un dominio (e dunque avente per definizione un elemento minimo, corrispondente alla funzione mai definita $\lambda s : S.\perp$), tale requisito viene soddisfatto nella categoria degli insiemi parzialmente ordinati e completi **CPO**. Inoltre, è immediato verificare che tale categoria soddisfa tutti i requisiti richiesti per l'interpretazione dei termini. Essa rappresenta dunque un esempio di categoria concreta all'interno del quale è possibile interpretare i termini di $\lambda_C(\Sigma)$.

4.5 L'interpretazione

Il passo finale richiesto per esprimere una semantica denotazionale per **TinyC** consiste nell'interpretare i termini del metalinguaggio all'interno della categoria. Considerata la categoria **CPO**, avente N come **nno**, un oggetto V per le variabili e un oggetto terminale 1, e all'interno del quale è presente la monade $TA = (A \times S)^S_\perp$, con $S \triangleq N^V$, l'interpretazione dei tipi avviene come segue:

- $[\![nat]\!] = N$
- $[\![1]\!] = 1$
- $[\![A \to B]\!] = [\![B]\!]^{[\![A]\!]}$

- $[\![TA]\!] = T[\![A]\!] = ([\![A]\!] \times S)^S_\perp$

Come richiesto da quanto espresso in 2.2.2.

L'interpretazione di un termine M del metalinguaggio, dovrà soddisfare la proprietà $M : A \Rightarrow [\![M]\!] : [\![A]\!]$.

I costrutti *let* e *val* sono facilmente interpretabili. La monade $TA = (A \times S)^S_\perp$ può essere infatti riscritta come $(\mathcal{S}(\_)_\perp)A$, dove $\mathcal{F}$ è il trasformatore di monadi dei side effects dell'equazione 3.7.1, e $(\_)_\perp$ è la monade del lifting presentata in sezione 3.3. Dalle equazioni 3.7.2 e 3.7.3 si ha dunque:

$$(val_{\mathcal{S}(\cdot)_\perp}\ [\![M]\!])s = val_\perp < [\![M]\!],s > \quad (4.5.1)$$

$$(let_{\mathcal{S}(\cdot)_\perp}\ x = [\![M]\!]\ in\ [\![N]\!])s = let_\perp\ < x,s' >= [\![M]\!]s\ in\ [\![N]\!]s' \quad (4.5.2)$$

Per ognuna delle costanti e delle primitive computazionali introdotte all'interno del metalinguaggio, dovrà essere definita un'interpretazione all'interno di **CPO**.
Per i termini che restituiscono un valore di tipo *nat* nel metalinguaggio, si dovrà ottenere un'interpretazione nell'oggetto N. L'interpretazione è semplice da definire:

$$[\![n]\!] = \mathrm{n} \quad (4.5.3)$$

$$[\![\underline{\theta}(m,n)]\!] = [\![m]\!]\ \theta\ [\![n]\!] \quad (4.5.4)$$

$$(4.5.5)$$

L'interpretazione del termine $* : 1$ avverrà nell'unico elemento dell'oggetto terminale $[\![1]\!]$ di **CPO**.
Infine, per ognuna delle primitive computazionali introdotte nel metalinguaggio, si dovrà esibire un'opportuna interpretazione.

La primitiva $read_x : Tnat$ dovrà essere interpretata in $(N_\perp \times S)^S_\perp$. Difatti, presa in input una configurazione di memoria (ovvero una funzione da variabili in naturali), dovrà restituire il naturale associato a tale variabile, senza mutare il contenuto della configurazione di memoria. L'equazione che descrive tale effetto è data da:

$$[\![read_x]\!]s = val_\perp\ (< s(x),s >) \quad (4.5.6)$$

La primitiva $upd_x : nat \to T1$ prenderà in input un numero naturale, e restituirà una funzione che ha per dominio l'insieme delle configurazioni di memoria, e per codominio un elemento dell'insieme $(1 \times S)_\perp$. L'effetto di tale primitiva sarà quello di aggiornare la configurazione di memoria s con il valore n passato in input alla primitiva upd_x.

$$[\![upd_x\ M]\!]s = val_\perp < *,s[x \leftarrow [\![M]\!]] > \quad (4.5.7)$$

Per interpretare una primitiva del tipo $cond_A(n,M,N)$, sarà necessario effettuare case analysis sull'interpretazione $[\![n]\!]$ di n. Si ottiene:

$$\begin{aligned}[\![cond_A(N,M,L)]\!]s &= case\ [\![L]\!]\ of \\ &\quad 0.[\![M]\!]s \\ &\quad s(n).[\![L]\!]s\end{aligned}$$

L'interpretazione degli operatori di astrazione e composizione è di ovvia definizione:

$$[\![fn\ x : A \Rightarrow M]\!] = \lambda x : [\![A]\!].[\![M]\!]_{x:A} \tag{4.5.8}$$

$$[\![MN]\!] = [\![M]\!]([\![N]\!]) \tag{4.5.9}$$

Infine, si è assunto che l'operatore di punto fisso sia definito per il solo tipo $T1$. Ciò permette di interpretare tale operatore come:

$$[\![fixM]\!] = \mu x.[\![M]\!]_{x:T1} \tag{4.5.10}$$

dove, assumendo $x \in FV(M)$, $\mu x.[\![M]\!]$ denota il minimo elemento x in $(1 \times S)^S_\perp$ isomorfo a $[\![M]\!]_{x:T1}x$.
Essendo $T1 = (1 \times S)^S_\perp$ un dominio, difatti, esiste sempre per un termine $M : T1 \to T1$ l'elemento $\mu x.[\![M]\!]x$. Si noti tuttavia che non è garantito che tale elemento sia effettivamente calcolabile[1].
Le equazioni qui presentate definiscono l'interpretazione dei termini del metalinguaggio nella categoria dei CPO. E' immediato veificare la correttezza dell'interpretazione

Teorema 4.5.1 (Correttezza)**.** $\forall M,N \in \lambda_C\Sigma.M = N \Rightarrow [\![M]\!] = [\![N]\!]$

Dimostrazione. Per dimostrare l'enunciato, è sufficiente mostrare che le uguaglianze di termini assunte nella sezione 4.2 risultano valide anche per le rispettive interpretazione dei termini.

4.2.1 $cond_A(0,M,N) = M$

$$\begin{aligned}[\![cond_A(0,M,N)]\!]s &= case\ [\![0]\!]s\ of \\ &\quad z.[\![M]\!]s \\ &\quad s(n).[\![N]\!]s \\ \\ &= [\![M]\!]s\end{aligned}$$

4.2.2 $cond_A(n,M,N) = N$, con $n \neq 0$. La dimostrazione è analoga a quella effettuata per il caso 4.2.1.

[1] Dalla teoria della calcolabilità, difatti, se si potesse sempre calcolare,per una funzione $f : T1 \to T1$, l'elemento $\mu x.f(x) \in T1$, si otterrebbe una soluzione al problema dell'arresto

4.2.3 $(fn\ x : A \Rightarrow M)N = M\{N/x\}$.

$$\begin{aligned}[\![(fn\ x : A \Rightarrow M)N = M\{N/x\}]\!] &= (\lambda x : [\![A]\!].[\![M]\!]_{x:A})([\![N]\!]) \\ &= [\![M\{N/x\}]\!]^2\end{aligned}$$

4.2.4 $M\ fix\ M = fix\ M$, con $M : T1 \to T1$

$$\begin{aligned}[\![M\ fix\ M]\!] &= [\![M]\!](\mu x.[\![M]\!]) \\ &= \mu x.[\![M]\!] \\ &= [\![fix\ M]\!]\end{aligned}$$

□

4.6 Interpretazione dei comandi TinyC

Definite la traduzione $(\!|\ \cdot\ |\!)$ dei comandi di **TinyC** all'interno del metalinguaggio $\lambda_C(\Sigma)$ e l'interpretazione $[\![\cdot]\!]$ dei termini di quest'ultimo all'interno di una categoria concreta, l'interpretazione di un comando C corrisponderà a $[\![(\!|C|\!)]\!]$. Con abuso di notazione, si ometteranno le parentesi corrispondenti alla traduzione dei comandi nel metalinguaggio. Per ricavare l'interpretazione dei comandi è spesso necessario affrontare una lunga sequenza di calcoli, sebbene essi non siano complicati. Per semplicità, si è preferito omettere tali calcoli e presentare direttamente la forma più ridotta delle equazioni corrispondenti all'interpretazione dei rispettivi comandi. Inoltre, per semplificare ulteriormente tali equazioni, si farà uso del seguente risultato:

Lemma 4.6.1. *Indicato con $\mathscr{E}$ l'insieme delle espressioni di **TinyC**, sia $v : \mathscr{E} \times S \to N$ così definita:*

$$\begin{aligned}v(n,s) &= n \\ v(x,s) &= s(x) \\ v(E_1\ \theta\ E_2,s) &= v(E_1,s)\ \theta\ v(E_2,s)\end{aligned}$$

Allora $[\![E]\!]s = val_\perp\ < v(E,s),s >$.

Dimostrazione. Per induzione sulla struttura delle espressioni. Per i casi base, n ed x, la prova è immediata. Sia dunque $\mathrm{E} \equiv \mathrm{E}_1\ \theta \mathrm{E}_2$: scrivendo l'interpretazione di tale espressione si ha

$$\begin{aligned}[\![\mathrm{E}_1\ \theta \mathrm{E}_2]\!]s &= let_\perp\ < m,s' >= [\![\mathrm{E}_1]\!]s\ in \\ &\quad\ let_\perp\ < n,s'' >= [\![\mathrm{E}_2]\!]s'\ in \\ &\quad\ val_\perp < m\ \theta\ n,s'' >\end{aligned}$$

Per ipotesi induttiva, $[\![\mathrm{E}_1]\!]s = val_\perp \ < v(\mathrm{E}_1,s),s >$, $[\![\mathrm{E}_2]\!] = val_\perp \ < v(\mathrm{E}_2,s'),s' >$. Si ottiene dunque $s'' = s' = s$, e l'intera equazione viene ridotta a:

$$\begin{aligned} [\![\mathrm{E}_1 \ \theta \mathrm{E}_2]\!]s &= val_\perp \ < v(\mathrm{E}_1,s) \ \theta v(\mathrm{E}_2,s),s > \\ &= val_\perp \ < v(\mathrm{E}_1 \ \theta \ \mathrm{E}_2,s),s > \end{aligned}$$

□

Sfruttando tale risultato, l'interpretazione dei comandi di **TinyC** può essere esplicitata come segue:

$$[\![\mathrm{skip}]\!]s = val_\perp \ < *,s > \tag{4.6.1}$$

$$[\![\mathrm{x:=E}]\!]s = val_\perp \ < *,s[x \leftarrow v(E,s)] > \tag{4.6.2}$$

$$[\![\mathrm{C}_1 \ ; \ \mathrm{C}_2]\!]s = let_\perp \ < x,s' >= [\![\mathrm{C}_1]\!]s \ in \ [\![\mathrm{C}_2]\!]s' \tag{4.6.3}$$

$$\begin{aligned} [\![\mathrm{if \ E \ then \ C}_1 \ \mathrm{else \ C}_2]\!]s = \ & case \ v(\mathrm{E},s) \ of \\ & z.[\![\mathrm{C}_2]\!]s \\ & succ(n).[\![\mathrm{C}_1]\!]s \end{aligned} \tag{4.6.4}$$

$$\begin{aligned} [\![\mathrm{while \ E \ do \ C}]\!]s = \ & (\mu f.\lambda s : S.case \ v(\mathrm{E},s) \ of \\ & z.val_\perp < *,s > \\ & succ(n).let_\perp \ < x,s' >= [\![\mathrm{C}]\!]s \ in \ fs')s \end{aligned} \tag{4.6.5}$$

5

Monadi nella Language Based Security

In questo capitolo verranno presentate alcune definizioni di base relative alla sicurezza dei sistemi informatici e alla descrizione delle politiche e dei meccanismi di sicurezza. In particolare, si considererà il problema dell'information flow, in cui la memoria viene partizionata in due zone, dette **memoria alta** e **memoria bassa**; la politica di sicurezza stabilisce che non deve essere permesso alcun flusso di informazione (anche parziale) dalla memoria alta alla memoria bassa. Verranno dunque esposte alcune soluzioni, basate sia su approcci tradizionali ([SM03], [VSI96], [SV98]) che su tecniche sviluppate più recentemente ([CC08], [HH05], [VBC+04]), opportunamente selezionate tra quelle esistenti nella letteratura della **Language Based Security**. L'obiettivo che ci si pone consiste nel riuscire a definire una semantica denotazionale per il linguaggio **TinyC** che, differentemente da quella esibita in 4, preveda uno stato di errore quale risultato di una computazione: per tale semantica si mostrerà che l'interpretazione di un'esecuzione (ovvero dall'interpretazione di un programma a cui viene applicata una configurazione di memoria iniziale) che viola la politica dell'information flow corrisponderà allo stato di errore introdotto.
La semantica denotazionale che cattura il concetto di sicurezza verrà definita utilizzando il λ-calcolo computazionale e le monadi. Utilizzando questo approccio, si è notato che, per descrivere il comportamento dei programmi di **TinyC** rispettivamente alla politica di information flow, è necessario associare diversi effetti computazionali alle espressioni e ai programmi: un programma agisce difatti sulle configurazioni di memoria, e può provocare un flusso di informazioni dalla memoria alta alla memoria bassa. Differentemente, le espressioni vengono utilizzate per calcolare un valore a partire da una singola configurazione di memoria: da questo punto di vista la valutazione di un'espressione non può provocare di per sè un flusso di informazioni, tuttavia ad ogni espressione dovrà essere associato un **livello di sicurezza** che denoti se il valore calcolato dall'espressione contenga dati che non devono essere copiati nella memoria bassa.

5.1 Definizioni realtive alla sicurezza

La sicurezza di un sistema informatico viene definita a partire da un insieme di proprietà che tale sistema deve soddisfarre. Allo stato attuale dell'arte, vengono universalmente identificate le seguenti come proprietà che debbono essere soddisfatte per garantire la sicurezza di un sistema:

Confidenzialità: Le informazioni presenti all'interno del sistema non devono poter essere accedute da chi non dispone dei privilegi necessari.

Integrità: Le informazioni del sistema non devono poter essere alterate da chi non ha i privilegi necessari per eseguire tale operazione. Inoltre, non deve essere possibile alterare la fonte di origine di tali informazioni.

Disponibilità: Le informazioni presenti all'interno del sistema devono poter essere accedute da chiunque abbia i privilegi necessari per venire a conoscenza di tali informazioni.

Nell'ambito della sicurezza informatica, viene effettuata una distinzione tra la **politica di sicurezza** che si vuole soddisfare, e i **meccanismi di sicurezza** utilizzati per assicurare che una data politica di sicurezza venga rispettata.

In [Bis03], un sistema informatico viene caratterizzato come un insieme di stati rappresentanti specifiche informazioni relative alla sicurezza. Inoltre, è possibile specificare come avvengano le transizioni tra stati di sicurezza in base alle azioni eseguite su tale sistema. Una politica di sicurezza viene identificata con un partizionamento di tale insieme di stati in due sottoinsiemi: una partizione conterrà gli stati reputati **sicuri** rispetto alla politica di sicurezza, mentre gli stati rimanenti vengono classificati come **insicuri**, ovvero che violano la politica di sicurezza.

Un meccanismo di sicurezza viene inteso come una parte di software il cui scopo è, data una politica di sicurezza, evitare che il sistema effettui una transizione da uno stato sicuro ad uno stato insicuro. Esistono tuttavia delle politiche per le quali non è possibile trovare un meccanismo di sicurezza che soddisfi tale requisito: in tali casi, lo scopo dei meccanismi di sicurezza è quello di riuscire a rilevare le violazioni della politica. Dal punto di vista formale, siano

- Q un insieme, eventualmente infinito, di stati di sicurezza del sistema.
- Σ un insieme di azioni consentite all'interno del sistema, o **Alfabeto**.
- P l'insieme degli stati di sicurezza sicuri, rispetto ad una data politica di sicurezza.
- $\delta : Q \times \Sigma \to Q$ una funzione di transizione.

Ad ogni meccanismo di sicurezza viene associato dunque un insieme di stati $M \subseteq Q$, coincidenti con l'insieme di stati reputati sicuri dal meccanismo di sicurezza. É possibile classificare i meccanismi di sicurezza in 3 tipologie, in base all'insieme di stati M che rappresentano tale meccanismo:

Strict: $M \subset P$. Il meccanismo di sicurezza non permette di transitare in degli stati non sicuri. Tuttavia, esistono degli stati sicuri che non vengono però permessi dal meccanismo di sicurezza.

Exact: $M = P$. Gli stati permessi dal meccanismo di sicurezza coincidono con tutti e soli gli stati sicuri secondo la descrizione della politica di sicurezza.

Broad: $P \subset M$. Il meccanismo di sicurezza permette ad un sistema di trovarsi in tutti gli stati sicuri secondo la definizione della politica di sicurezza. Tuttavia, esistono anche degli stati insicuri che vengono ammessi dal meccanismo di sicurezza.

Dal punto di vista della **Language Based Security**, è preferibile definire una politica di sicurezza come una proprietà logica che deve essere soddisfatta dal sistema. In ogni caso, Data una proprietà P di sicurezza, è comunque possibile classificare un meccanismo di sicurezza all'interno dei sottoinsiemi **Strict**, **Exact** e **Broad**.

Il problema dell'Information Flow

La Language Based Security è un area dell'informatica che si occupa della definizione di politiche di sicurezza e dell'implementazione di meccanismi di sicurezza per tali politiche orientate ai linguaggi di programmazione.

Uno dei problemi più noti nella letteratura della language based security è dato dalla rilevazione degli **Information Flows**.
Dato un linguaggio di programmazione imperativo, la memoria viene partizionata in una zona detta **alta**, contenenti informazioni segrete e in una zona detta **bassa**. La politica di sicurezza stabilisce che, al termine dell'esecuzione di un programma, nessuna informazione, seppure parziale, deve essere stata copiata dalla zona di memoria alta alla zona di memoria bassa. Tale politica di sicurezza rientra dunque nell'insieme delle politiche concernenti la confidenzialità.

Una variabile $x \in \mathcal{V}$ di un programma è detta essere una variabile **Low** se, e solo se, essa appartiene alla zona di memoria bassa. Differentemente, tale variabile verrà detta **High**. Una variabile bassa l viene denotata con $l : L$; analogamente, una variabile alta h viene denotata con $h : H$. Come esempio di linguaggio di programmazione verrà considerato **TinyC**, di cui è già stata osservata la **Turing Completezza**. Un programma di **TinyC** verrà indicato con la dicitura **comando**.

Il problema dell'information flow può essere descritto formalmente definendo una relazione di **Low equivalenza** sugli stati della memoria, indotta dalla generica relazione di equivalenza su tale insieme.

Definizione 5.1.1.

Low equivalenza

Date due configurazioni di memoria s_1,$s_2 : \mathcal{V} \rightarrow N$*, esse sono dette essere low equivalenti* ($s_1 =_L s_2$) *se, e solo se*

$$\forall x.x : L \Rightarrow s_1(x) = s_2(x) \tag{5.1.1}$$

Si osservi che, dato un comando C, se per ogni sua possibile esecuzione non è possibile rilevare un flusso di informazioni, allora tale comando si comporterà in maniera analoga su quelle configurazioni di memoria tra loro low equivalenti. Tale principio prende il nome di non interferenza:

Definizione 5.1.2.

Non interferenza soggetta a terminazione

Sia dato un comando C, e siano s_1,s_2 *due stati low equivalenti tali che* $[\![C]\!]s_1 = val_\perp \ < *,s_1' >$, $[\![C]\!]s_2 = val_\perp < *,s_2' >$. *Se* $s_1' \neq_L s_2'$, *allora il comando è detto* ***interferente****. Diversamente, se non è mai possibile trovare tale coppia di stati, allora il comando viene detto* ***non interferente****.*

Un comando viene quindi definito sicuro rispetto alla politica dell'information flow se e solo se esso soddisfa la proprietà di non interferenza.

Prima di analizzare le tecniche utilizzate in letteratura per rilevare i comandi che permettono flussi di informazione, si vuole mostrare che per tale problema non esistono dei meccanismi di sicurezza esatti. Come conseguenza, poiché è di scarso interesse implementare dei meccanismi estesi per la rilevazione dei flussi di informazione, tutte le tecniche che verranno analizzate portano all'implementazione di meccanismi di sicurezza severi.

Teorema 5.1.1 ([DD97])**.** *Non esiste un meccanismo di sicurezza esatto per il problema dell'information flow.*

Dimostrazione. Si supponga che tale meccanismo esista. Dunque, esiste un algoritmo $\mathcal{A}$ che accetta se, e solo se, dato in input un comando, esso è non interferente.
Dato un comando C, si supponga inoltre che ogni variabile di tale comando sia low. Infine, si introducano due nuove variabili non presenti in C, $l : L$ e $h : H$. Se si passa in

input ad $\mathcal{A}$ il comando C; l:=h;, esso accetterà se, e solo se, il comando C termina la sua esecuzione. Difatti, un flusso di informazione in tale comando avviene solamente in seguito all'esecuzione del comando l:=h. Dalla Turing Completezza di **TinyC**, e dall'impossibilità di decidere il problema della fermata, si ottiene che l'algoritmo $\mathcal{A}$ non può esistere. □

L'impossibilità di riuscire a rilevare tutti e soli i flussi di informazione è data dalla possibilità di ottenere, in un comando, dei flussi di informazione impliciti.
Un flusso di informazione esplicito corrisponde ad un comando del tipo $l := h$, dove $l : L$,$h : H$. Tuttavia, se si considera il comando

```
1 if h
2     then l:=1
3     else l:=0
```

è immediato notare che esso costituisce un flusso di informazioni **implicito**. Difatti, pur non essendo presente un assegnamento diretto che provoca il flusso di informazioni, è possibile ottenere dell'informazione sulla variabile alta h, in base al valore contenuto in l al termine dell'esecuzione del comando. In letteratura, i flussi di informazione impliciti vengono anche detti **covert channels**.
Infine, dati due comandi C_1 e C_2, l'interferenza di C_1 non implica l'interferenza di $C_1; C_2$, come mostrato dal seguente esempio:

```
1 l := h;
2 l := 0;
```

Nonostante il comando $l := h$ sia interferente, non lo è l'intero comando illustrato.

5.2 Lavoro correlato

5.2.1 Analisi statica

Le tecniche più utilizzate, nell'approcciare il problema del flusso di informazioni, consistono nell'analisi statica. Dato un tipaggio delle variabili nei due tipi **L** e **H**, vengono definiti dei sistemi che garantiscono la possibilità di tipare un programma solamente se questi è non interferente. Tuttavia, come conseguenza del teorema 5.1.1 esistono dei comandi non interferenti che non possono essere tipati tramite il sistema di tipi utilizzato.
Al meglio della mia conoscenza, il primo lavoro basato sui sistemi di tipi è dato da [VSI96]. Il risultato di tale lavoro è stato poi migliorato, da parte degli autori, estendendo il sistema di tipi ad un linguaggio che prevede la possibilità di eseguire comandi in parallelo, [SV98].
Uno dei vantaggi derivanti dalle tecniche di analisi statica è dato dalla possibilità di verificare la non interferenza di un comando a tempo di compilazione, senza necessità di effettuare ulteriori controlli a tempo di esecuzione. Tuttavia, da questa osservazione deriva la possibilità di reputare corretti solamente quei comandi per cui in ogni possibile esecuzione viene garantita la mancanza di flusso di informazione (in accordo con la definizione 5.1.2).
Inoltre, uno studio accurato illustrato in [HMS96], mostra che dei meccanismi di sicurezza esatti tramite analisi statica possono essere implementati solamente per quelle politiche

descritte da proprietà decidibili: difatti, questa rappresenta una enorme limitazione al potere computazionale dei meccanismi basati sull'analisi statica.
Tuttavia, in [CHM05], i limiti computazionali dell'analisi statica vengono aggirati ricorrendo a strumenti di base di **teoria dell'informazione**. In tale lavoro viene definito un sistema probabilistico (e pertanto non soggetto alle limitazioni delle tecniche di analisi statica) in cui viene garantita la rilevazione di comandi interferenti con alta probabilità. Il meccanismo derivante da tale sistema di tipi è dunque esteso, in quanto permette (seppure con probabilità trascurabile) il tipaggio di comandi interferenti.

5.2.2 Controllo a runtime e Riscrittura di Programmi

Negli ultimi anni sono stati sviluppati nuovi approcci al problema dell'information flow, che prendono in considerazione la possibilità di rilevare un flusso di informazione durante l'esecuzione di un programma, eventualmente sviluppando una fase intermedia di compilazione del codice, introducendo all'interno dello stesso delle annotazioni. Tali approcci sono basati su due differenti tecniche, entrambe discusse in [HMS96]:

Execution Monitoring: I controlli vengono effettuati a tempo di esecuzione da una parte di software detta **Execution Monitor**. Compito di tale componente è quello di memorizzare lo stato di sicurezza del sistema, interrompendo l'esecuzione di un comando nel caso venga riscontrata una violazione della politica di sicurezza.

Riscrittura di programmi: In questo approccio la violazione di una politica di sicurezza viene affrontata riscrivendo il codice che provoca tale violazione con un codice equivalente, ma che risulti essere sicuro. Le tecniche di riscrittura di programmi spesso introducono delle annotazioni di codice a livello di compilazione, che verranno utilizzate per effettuare i controlli a tempo di esecuzione.

Uno dei maggiori lavori effettuati, utilizzando le tecniche di riscrittura dei programmi, è dato da [VBC+04]. **RIFLE** costituisce un framework per la rilevazione a tempo di esecuzione dei flussi di informazione. In fase di compilazione, i covert channel vengono convertiti in flussi di informazione espliciti, rendendo così necessaria la sola rilevazione di questi ultimi durante l'esecuzione dei programmi.

Dal punto di vista dell'execution monitoring, l'unico lavoro conosciuto è dato da [CC08]. L'approccio utilizzato è quello di sfruttare la dipendenza che intercorre tra le variabili durante l'esecuzione di un comando, rappresentandola tramite opportune strutture che prendono il nome di **grafi di dipendenza**. L'articolo mostra inoltre come i meccanismi di sicurezza presentati siano scalabili a linguaggi di programmazione reali.
Tuttavia, sebbene l'uso dei grafi di dipendenza permette di rilevare una discreta quantità di covert channel, gli autori hanno proposto un modello leggermente diverso da quello approcciato classicamente in letteratura. Il modello computazionale utilizzato, difatti, presenta una primitiva *output* x, corrispondente all'unico evento per il quale debbono essere eseguiti i controlli sulla dipendenza delle variabili.

Infine, in [HH05], viene presentato un approccio al problema dell'information flow basato sulla teoria delle categorie e sulle monadi. L'articolo introduce delle strutture nominate **separation kernels** per il rilevamento dei flussi di informazione. Tuttavia, il lavoro eseguito non costituisce una reale soluzione al problema. Per ottenere una soluzione tramite l'uso delle monadi, viene infatti introdotto il concetto di **Non Interferenza Atomica**: identificando con l i comandi che agiscono sulla memoria bassa, e con h i comandi che agiscono sulla memoria alta, un comando è atomicamente non interferente se, e solo se,

$$l; h =_L h; l \tag{5.2.1}$$

La proprietà 5.2.1 non coincide con la definizione 5.1.2. Si consideri infatti il comando

```
1 h := l;
2 h := h + 5;
3 l := h;
```

Tale programma, in base alla definizione 5.1.2, non è interferente. Tuttavia, esso è atomicamente interferente, in quanto il comando ottenuto permutando gli assegnamenti delle righe di codice *2* e *3*

```
1 h:=l;
2 l := h;
3 h := h +5;
```

Non produce la stessa configurazione di memoria del programma originario.

5.3 Presentazione del lavoro

Il lavoro che si presenterà nel seguito di questo capitolo è stato ispirato, nella maggior parte, da [VSI96] e [CC08]. Si sfrutteranno infatti le idee presenti nel sistema di tipi di Volpano et al. per ottenere una distinzione tra i contesti di sicurezza in cui vengono eseguiti i comandi, mentre si utilizzerà l'idea di Cavadini e Cheda nello sfruttare le dipendenze di variabili per analizzare i flussi di informazione tramite l'uso delle monadi e del λ-calcolo computazionale.
L'obiettivo che ci si pone non è quello di sviluppare nuove tecniche per la rilevazione di flussi di informazione durante l'esecuzione dei programmi, bensì quello di definire una semantica che tenga conto di tali flussi quali effetti computazionali. Le equazioni che verranno ottenute per i comandi **TinyC** possono essere intese come un **incremento** delle equazioni esibite in 4.6, dove l'incremento è dato dall'introduzione dello stato di errore nel modellare l'effetto computazionale associato ai comandi: il teorema **??** che verrà mostrato in seguito mostra infatti che esiste una relazione tra l'interpretazione di un comando secondo la semantica di **??** e tra quella che si definirà in questo capitolo.

5.4 Contesti di Sicurezza

I contesti di sicurezza rappresentano un concetto centrale nello sviluppo di una semantica per i flussi di informazione. Nel capitolo 4 si è definita la semantica per un linguaggio

di programmazione imperativo. Nel metalinguaggio utilizzato per definire tale interpretazione, sono stati individuate le seguenti primitive computazionali, rispettivamente per la lettura e l'aggiornamento delle variabili:

- $read_x$ per ogni variabile x.

- $upd_x\ n$ per ogni variabile x.

L'aggiornamento di una variabile, dal punto di vista della sicurezza, produrrà tuttavia degli effetti diversi in base alla zona di memoria dei dati presenti nell'espressione utilizzata per l'aggiornamento. Si consideri, ad esempio, il comando

```
1 l := h
```

Tale comando costituisce un flusso di informazione esplicito, in quanto il valore della variabile bassa l viene aggiornato con il contenuto della variabile alta h. In questo caso, si dirà che l'aggiornamento della variabile l avviene in un **contesto di sicurezza alto**, in quanto i dati utilizzati per l'aggiornamento della variabile *dipendono* da una variabile alta. Analogamente, l'esecuzione di un comando potrà produrre un risultato differente a seconda del contesto in cui esso viene eseguito. Si consideri ad esempio il comando:

```
1 l := 0
```

Tale comando è non interferente, in quanto non può copiare delle informazioni contenute in una variabile alta. Si consideri però il seguente comando:

```
1 if  h then
2       skip
3 else
4     l := 0
```

Tale comando costituisce un flusso di informazione implicito, dal momento che riesce a copiare nella variabile bassa l dell'informazione parziale sulla variabile alta. In questo caso l'interferenza è data dal comando $l := 0$, che viene eseguito all'interno di uno statement condizionato da una variabile alta, detta anche **high guard**. Il valore utilizzato per aggiornare l, in questo caso, dipende dalla variabile alta h, e pertanto l'aggiornamento della variabile avviene in un contesto di sicurezza alto.

Formalmente, viene definito un insieme di contesti di sicurezza $Q = \{L,H\}$. Inoltre, viene definita una funzione $\tau : \mathcal{V} \to Q$ che associa ad ogni variabile il proprio livello di sicurezza. In questo modo sarà possibile calcolare, algoritmicamente, il contesto di sicurezza in cui vengono eseguite le operazioni di aggiornamento delle variabili. Al momento di definire l'interpretazione dei comandi, si mostrerà come tale interpretazione sia differente a seconda del contesto di sicurezza utilizzato.

5.5 Cenni sull'interpretazione

Si è accennato, seppure in maniera descrittiva, che l'interpretazione dei comandi è sensibile al contesto di sicurezza utilizzato per l'interpretazione dello stesso. Differentemente,

dal punto di vista della sicurezza, la valutazione delle espressioni corrisponde a generare il contesto di sicurezza in cui dovranno essere interpretati i comandi seguenti.

Un'espressione di **TinyC** viene intesa dunque ora come una funzione il cui dominio corrisponde all'insieme $S = \mathcal{V} \to N$, e il cui risultato è dato da una coppia $< n,q >$, dove $n \in N$,$q \in Q$. Nella semantica che si vuole definire, dunque, si richiederà che l'interpretazione di un'espressione corrisponda ad un elemento dell'oggetto $(N \times Q)^S$. Al momento di definire l'interpretazione in maniera formale, verrà mostrato come si possa ottenere un'interpretazione delle espressioni in tale oggetto ricorrendo alla monade dei *context readers* $T_E\ A = A^S$, introdotta in 3.8.

L'interpretazione di un comando corrisponde ad una funzione che, presi in input un contesto di sicurezza e uno stato di configurazione della memoria, restituisce uno tra i seguenti possibili risultati:

- un elemento dell'insieme $1 \times S$, ovvero una configurazione di memoria di output.
- l'elemento $*$ dell'oggetto 1, corrispondente ad una situazione di errore.
- l'elemento $\perp$, corrispondente alla non terminazione.

La monade che verrà utilizzata per interpretare i comandi è data dunque da:

$$T_C\ A = (((A \times S) + 1)^S_\perp)^Q$$

La proprietà che si vuole ottenere dalla semantica che verrà definita è data dall'interpretare i comandi interferenti nello stato di errore $* \in 1$.

5.6 Presentazione del metalinguaggio

Il linguaggio concreto per il quale si vuole definire una semantica che consideri il flusso di informazioni è dato da **TinyC**, presentato in 4.1. Si è accennato che si vogliono differenziare gli effetti computazionali prodotti dalle espressioni e dai comandi: il metalinguaggio che verrà utilizzato per la traduzione sarà differente da quello mostrato in 4.2. Devono infatti essere introdotti i tipi necessari per rappresentare i differenti effetti computazionali, ed è necessario effettuare un'analisi sui tipi delle singole primitive computazionali che verranno introdotte.

Per gli scopi che si sono prefissi, ad ogni espressione verrà associato, oltre che un valore nei naturali, anche un livello di sicurezza associato a tale espressione. Per tale motivo, abbiamo preferito indicare le costanti computazionali del tipo n, che nel metalinguaggio di 4 avevano tipo *nat*, con il tipo *value*: tale scelta è stata effettuata per enfatizzare il concetto che tali costanti non verranno interpretate nell'insieme dei naturali, ma bensì in un **valore** scelto nell'insieme $N \times Q$, dove Q è l'insieme dei contesti di sicurezza; inoltre, tra le costanti computazionali si avrà anche $* : 1$. Oltre ai tipi freccia, devono essere

introdotti i costruttori di tipi corrispondenti ai diversi effetti computazionali che vogliono essere considerati nel metalinguaggio. Nella sezione 5.5, sono stati individuati due tipi computazionali differenti: uno per le espressioni T_E e uno per i comandi T_C. Un tipo, nel metalinguaggio che verrà definito, è dato dunque da un termine della seguente **BNF**:

$$\tau \ ::= \ value \mid 1 \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2 \mid T_E\ \tau \mid T_C\ \tau \tag{5.6.1}$$

Le primitive e le costanti computazionali che vengono introdotte all'interno del linguaggio corrispondono a quelle definite in 4.2. I tipi di tali termini, tuttavia, devono essere ridefiniti in accordo con i nuovi effetti computazionali introdotti.

- Costanti computazionali: tra queste individuiamo le costanti $n : value$, $* : 1$.
- Gli operatori binari $\underline{\theta}$ devono restituire, oltre che il naturale ottenuto dall'applicazione dell'operatore agli operandi, anche il livello di sicurezza ottenuto come risultato dell'operazione. Tale livello viene definito a partire dai livelli di sicurezza degli operandi. Gli operatori $\underline{\theta}$ hanno dunque tipo $value \times value \to value$.
- La lettura di una variabile restituisce un valore associato a tale variabile. Si pone pertanto $read_x : T_E\ value$.
- L'effetto computazionale corrispondente all'aggiornamento di una variabile corrisponde alla restituzione di una configurazione di memoria o di uno stato di errore, come notato in 5.5. Sono stati inoltre identificati i termini del metalinguaggio che verranno interpretati in tale effetto computazionale come quei termini di tipo $T_C\ 1$. Infine, il risultato dell'aggiornamento dipenderà da un valore, costituito sia da un numero naturale che dal livello di sicurezza ad esso associato. Si ottiene $upd_x : value \to T_C\ 1$.

- Poiché sono stati introdotti due differenti costruttori di tipi, da associare ad effetti computazionali differenti, è necessario introdurre per ognuno di essi gli operatori *let* e *val*. Per il costruttore di tipi T_E verranno introdotti gli operatori let_E e val_E. Analogamente, per il costruttore T_C verranno utilizzati let_C e val_C.
- Così come è stato fatto per upd_x, il tipo dell'operatore di scelta dovrà tenere conto del livello di sicurezza del valore che verrà applicato a tale operatore. Si ottiene dunque $cond_A : value \times A \times A \to A$. Poiché il costrutto *cond* viene utilizzato per il controllo del flusso di esecuzione dei comandi, e poiché (come si vedrà in 5.7) i comandi saranno dati da termini del tipo $T_C\ 1$, per semplicità si è preferito introdurre tale operatore per i soli termini di tale tipo.
- L'operatore di punto fisso fix opera sui comandi, e pertanto si ottiene $fix : (T_C\ 1 \to T_C\ 1) \to T_C\ 1$.
- É necessario inoltre introdurre un operatore ε che muti i termini di tipo $T_E\ A$ nei termini di tipo $T_C\ A$. L'utilità di tale operatore risiede nel poter far interagire, tramite l'utilizzo di let_C e val_C, i termini a cui verranno associati effetti computazionali differenti.

- Infine, l'astrazione e l'applicazione di termini rimane immutata, in quanto a tali operatori non verranno associati degli effetti computazionali.

Il seguente sistema di tipi presenta matematicamente il metalinguaggio descritto:

Sistema di Tipi 5.6.1.

$$\frac{}{\Gamma \vdash * : 1} \qquad \frac{}{\Gamma \vdash n : value}$$

$$\frac{}{\Gamma \vdash read_x : T_E\ value} \qquad \frac{\Gamma \vdash m : value \quad \Gamma \vdash n : value}{\Gamma \vdash \underline{\theta}(m,n) : value}$$

$$\frac{\Gamma \vdash n : value}{\Gamma \vdash upd_x\ n : T_C\ 1} \qquad \frac{\Gamma \vdash n : value \quad \Gamma \vdash M : T_C\ 1 \quad \Gamma \vdash N : T_C\ 1}{\Gamma \vdash cond(n,M,N) : T_C\ 1}$$

$$\frac{\Gamma,x : A \vdash M : B}{\Gamma \vdash fn\ x : A \Rightarrow M : A \to B} \qquad \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$$

$$\frac{\Gamma \vdash M : T_C\ 1 \to T_C\ 1}{\Gamma \vdash fix(M) : T_C\ 1} \qquad \frac{\Gamma \vdash M : T_E\ A}{\Gamma \vdash \varepsilon\ M : T_C\ A}$$

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash val_E\ M : T_E\ A} \qquad \frac{\Gamma \vdash M : T_E\ A \quad \Gamma,x : A \vdash N : T_E\ B}{\Gamma \vdash let_E\ x = M\ in\ N : T_E\ B}$$

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash val_C\ M : T_C\ A} \qquad \frac{\Gamma \vdash M : T_C\ A \quad \Gamma,x : A \vdash N : T_C\ B}{\Gamma \vdash let_C\ x = M\ in\ N : T_C\ B}$$

Dove sono state omesse le regole strutturali, quali l'assioma $x : A \vdash x : A$ e il weakening, che si assume siano familiari al lettore.

5.7 Traduzione

Definito il metalinguaggio, il passo successivo, nel definire una semantica per il flusso di informazioni, consiste nel tradurre i comandi di **TinyC** nel metalinguaggio. Si è già osservato che si vogliono associare effetti computazionali ad espressioni e comandi, e che vengono utilizzati due costruttori di tipi differenti per i termini che verranno intepretati

in tali effetti computazionali. Si è preferito dunque dividere la traduzione in due parti, rispettivamente per la traduzione delle espressioni e dei comandi. Ancora una volta, verrà utilizzata la notazione $(\!| \cdot |\!)$ per indicare la traduzione di un comando o di un'espressione.

5.7.1 Traduzione delle espressioni

Dal punto di vista della sicurezza per i flussi di informazione, un'espressione corrisponde ad una funzione che, data una configurazione di memoria come argomento, restituisce una coppia $< n,q >$, dove n è il valore ottenuto dalla valutazione dell'espressione e q rappresenta il contesto di sicurezza associato ad essa. Per tali coppie è stato appositamente definito il tipo *value*. Sebbene tale aspetto debba essere considerato al momento di definire l'interpretazione dei termini, esso mette in evidenza come le espressioni debbano essere tradotte in termini del tipo T_E *value*.

Per le costanti, dunque, si ottiene la seguente traduzione:

$$(\!|\mathrm{n}|\!) = val_E\ n : T_E\ value \tag{5.7.1}$$

Se un'espressione coincide invece con una variabile x, la traduzione coinciderà con il termine $read_x$. Dalla definizione del termine $read_x$ in 5.6.1 si ha infatti che tale termine ha tipo T_E *value*. Il comportamento di tale operatore dal punto di vista semantico verrà considerato al momento di definire l'interpretazione:

$$(\!|\mathrm{x}|\!) = read_x : T_E\ value \tag{5.7.2}$$

Infine, un'espressione può essere data dalla composizione di due espressioni combinati tramite un operatore θ. Definendo $\mathrm{E} = \mathrm{E}_1\theta\mathrm{E}_2$, e supponendo di avere definito le traduzioni $(\!|\mathrm{E}_1|\!)$,$(\!|\mathrm{E}_2|\!) : T_E\ value$, sarà sufficiente valutare entrambe le espressioni ed applicare ai valori ottenuti l'operatore corrispondente $\underline{\theta}$ definito nel metalinguaggio.

$$(\!|\mathrm{E}_1\theta\mathrm{E}_2|\!) = let_E\ x = (\!|\mathrm{E}_1|\!)\ in\ let_E\ y = (\!|\mathrm{E}_2|\!)\ in\ \underline{\theta}(x,y) : T_E\ value \tag{5.7.3}$$

La traduzione fornita in questa soddisfa la proprietà di adeguatezza statica:

Proposizione 5.7.1 (Adeguatezza Statica per le Espressioni)**.** *Data un'espressione E di* ***TinyC****,* $(\!|E|\!) : T_E\ (value)$.

5.7.2 Traduzione dei comandi

Precedentemente si è stabilito di voler interpretare i comandi come computazioni che, dato un contesto di sicurezza ed una configurazione di memoria, restituiscono una memoria di output, oppure un errore oppure il simbolo $\perp$ associato alla non terminazione. Si è inoltre discusso di come i termini del metalinguaggio di tipo $T_C\ A$ verranno interpretati in oggetti rappresentanti tali effetti computazionali. Anche in questo caso, nel presentare la traduzione dei comandi, si daranno alcune intuizioni su come verrà definita l'interpretazione dei termini.

Si stabilisce dunque che i comandi di **TinyC** vengano tradotti in termini di tipo $T_C\ 1$.

Il caso più interessante, nella traduzione dei comandi, è dato dall'assegnamento. Si consideri un assegnamento del tipo

$$\text{x:= E}$$

Tale assegnamento coincide con il valutare l'espressione E, e quindi aggiornando il valore della variabile x con il valore valutato dall'espressione. Si noti inoltre che il contesto di sicurezza valutato dall'espressione E può condizionare l'aggiornamento di una variabile: tale eventualità è stata tenuta in considerazione nella definizione del sistema di tipi 5.6.1, in cui viene stabilito che i termini applicabili ad upd_x devono avere tipo $nat \times Q$.
Infine, un espressione viene tradotta in un termine di tipo $T_E\ value$; se si volesse tradurre l'assegnamento x := E nel metalinguaggio, non è possibile utilizzare il termine

$$let_C\ y = (\!|E|\!)\ in\ upd_x\ y$$

in quanto $(\!|E|\!) : T_E$ per la proposizione 5.7.1, e le regole di tipaggio definite in 5.6.1 stabiliscono che in $let_C\ x = M\ in\ N$ si deve avere $M : T_C\ A$, per un qualsiasi tipo A. Il problema che si presenta può essere risolto utilizzato l'operatore di conversione ε. Difatti, il seguente termine risulta tipabile nel metalinguaggio, e costituisce la traduzione degli assegnamenti:

$$(\!|\text{x:=E}|\!) = let_C\ y = \varepsilon(\!|\text{E}|\!)\ in\ upd_x\ y : T_C\ 1 \tag{5.7.4}$$

L'operatore ε verrà utilizzato in seguito ogni qualvolta sarà necessario convertire un termine di tipo $T_E\ A$ in un termine di tipo $T_C\ A$.

Il comando skip è semplice da tradurre, in quanto non produce un cambiamento della configurazione di memoria, nè può provocare un flusso di informazioni o generare nonterminazione. Si è scelto di tradurre skip come

$$(\!|\text{skip}|\!) = val_C * : T_C\ 1. \tag{5.7.5}$$

La traduzione della composizione di comandi C_1;C_2può essere ottenuta considerando che si dovrà procedere a valutare il comando C_1, e quindi a valutare il comando C_2 quando possibile. Assumendo che $(\!|\text{C}_1|\!)$,$(\!|\text{C}_2|\!)$ abbiano tipo $T_C\ 1$, si definisce

$$(\!|\text{C}_1;\text{C}_2|\!) = let\ x = (\!|\text{C}_1|\!)\ in\ (\!|\text{C}_2|\!) : T_C\ 1. \tag{5.7.6}$$

La traduzione di un comando del tipo if E then C_1 else C_2 può essere ottenuta valutando l'espressione E, e quindi utilizzando l'operatore $cond_{T_C\ 1}$ per selezionare il ramo dell'esecuzione desiderato. Si ottiene pertanto

$$(\!|\text{if E then C}_1\text{ else C}_2|\!) = let_C\ x = \varepsilon(\!|\text{E}|\!)\ in\ cond(x,(\!|\text{C}_1|\!),(\!|\text{C}_2|\!)) : T_C\ 1. \tag{5.7.7}$$

Infine, si deve definire la traduzione dei comandi del tipo while E do C. Si supponga che $(\!|\text{C}|\!) : T_C\ 1$. Ancora una volta si può notare che

$$\text{while E do C} = \text{if E then (C; while E do C;) else skip}$$

ed utilizzare l'operatore $fix : (T_C\ 1 \to T_C\ 1) \to T_C\ 1$ per tradurre tale comando. Senza ripetere l'intero ragionamento, analizzato già in 4.3, la traduzione del comando while E do C viene definita come

$$[\![\text{while E do C}]\!] = fix(fn\ f : T_C\ 1 \Rightarrow let_C\ x = \varepsilon[\![\text{E}]\!]\ in\ cond_{T_C\ 1}(x, let_C\ x = [\![\text{C}]\!] inf, val_C\ *)) \tag{5.7.8}$$

Così come per le espressioni, anche per i comandi si ottiene un risultato di adeguatezza:

Proposizione 5.7.2 (Adeguatezza Statica per i Comandi)**.** *Dato un comando C di* ***TinyC****,* $[\![C]\!] : T_C\ 1$.

5.8 Semantica della sicurezza

L'ultimo passo che si deve effettuare nel definire la semantica di **TinyC** dal punto di vista della sicurezza è dato dall'interpretazione dei termini del metalinguaggio in una categoria. A tale scopo, è necessario effettuare una discussione su quali siano i requisiti che la categoria scelta per l'interpretazione debba soddisfare. Inoltre, si dovrà mostrare che la trasformazione $T_C\ A = (((A \times S) + 1)^S_\perp)^Q$ è effettivamente una monade (tale ragionamento non deve essere ripetuto per $T_E\ A = A^S$, in quanto si è già introdotta la monade dei context readers in 3.8). Una volta che si sarà esibito un esempio di categoria concreta sul quale effettuare l'interpretazione dei termini, si potrà procedere a definire formalmente l'interpretazione dei termini del metalinguaggio negli oggetti di tale categoria.

5.8.1 Considerazioni sulla categoria

La presenza dei tipi freccia all'interno del metalinguaggio lascia intuire che l'interpretazione dovrà essere effettuata all'interno di una categoria chiusa. Dalle costanti computazionali introdotte, si deve inoltre assumere che esistano un **nno** N, un oggetto $\mathcal{V}$ che contenga le variabili, ed un oggetto Q utilizzato per i contesti di sicurezza. Poiché un termine di tipo *value* verrà interpretato in una coppia data da un numero naturale e da un valore, si deve inoltre assumere che esista l'oggetto $N \times Q$. Inoltre, al fine di poter calcolare il contesto di sicurezza delle espressioni, viene richiesto che su tale oggetto venga stabilita una relazione di ordinamento (nella fattispecie, si richiederà che $L < H$). L'esistenza dell'**nno** e dell'oggetto rappresentante i contesti di sicurezza si rendono necessari, in quanto si è osservato che il tipo *value* verrà interpretato in $(N \times Q)$. Dal momento che si è assunto di lavorare su una categoria chiusa, sarà possibile interpretare gli operatori di astrazione ed applicazione senza introdurre alcun altro vincolo sulla struttura della categoria.
Ancora una volta si porrà $S = \mathcal{V} \to N$, dove N.
Infine, si è scelto di interpretare i termini di tipo $T_C\ A$ all'interno dell'oggetto $((A \times S) + 1)^{SQ}_\perp$. Poiché in 5.6 sono stati introdotti i costrutti let_C e val_C, si devono inoltre esibire per essi delle interpretazioni che soddisfano gli assiomi del λ-calcolo computazionale, 2.2.3, 2.2.4 e 2.2.5. Ciò è possibile utilizzando i costruttori semantici introdotti in 3: difatti, tale monade è ottenibile tramite l'applicazione (nell'ordine) del costruttore del context reader 3.8 relativamente all'oggetto Q, del costruttore dei side effects 3.7, del costruttore degli

errori 3.6 e della monade del lifting 3.3. Formalmente, si ottiene infatti:

$$(\mathcal{Q}(\mathcal{S}(\mathcal{E}(\_)_\perp)))A = (((A \times S) + 1)^S_\perp)^Q \tag{5.8.1}$$

Dunque, la trasformazione di oggetti $T_C = ((A \times S) + 1)^{SQ}_\perp$ è una monade, per la quale gli operatori let_C e val_C possono essere ottenuti dalle definizioni date nelle rispettive sezioni.

La categoria **CPO**, utilizzata già in 4.4 per la definizione della semantica standard di **TinyC**, soddisfa i vincoli imposti nella presente discussione. Inoltre, poiché l'oggetto $((1 \times S) + 1)^{SQ}_\perp$ consiste di un dominio in tale categoria, è possibile dare l'interpretazione del costrutto $fix : (T_C\ 1 \to T_C\ 1) \to T_C\ 1$ facendo ricorso all'operatore di punto fisso.

Scelta dunque **CPO** come categoria per l'interpretazione dei termini, è possibile definire l'interpretazione dei tipi del metalinguaggio:

$$\begin{aligned}
[\![value]\!] &= (N \times Q) && (5.8.2)\\
[\![1]\!] &= 1 && (5.8.3)\\
[\![A \to B]\!] &= [\![B]\!]^{[\![A]\!]} && (5.8.4)\\
[\![T_E\ A]\!] &= [\![A]\!]^S && (5.8.5)\\
[\![T_C\ A]\!] &= ((([\![A]\!] \times S) + 1)^S_\perp)^Q && (5.8.6)
\end{aligned}$$

Nel seguito, per utilizzare una notazione scorrevole, si utilizzeranno $T_E\ A$ e $T_C\ A$ per indicare rispettivamente gli oggetti A^S e $((A \times S) + 1)^{SQ}_\perp$.

5.8.2 Interpretazione dei termini

Definita l'interpretazione dei tipi, si deve provvedere a fornire un'interpretazione dei termini in maniera tale che $M : A \Rightarrow [\![M]\!] \in [\![A]\!]$. Per ogni regola di inferenza del sistema di tipi 5.6.1 dovrà essere specificata l'interpretazione della conseguenza, assumendo di avere definita un'interpretazione per la premessa.

- $n : value$. In questo caso, è sufficiente porre

  $$[\![n]\!] =< n,L > \tag{5.8.7}$$

  In quanto si è già osservato che il livello di sicurezza associato ad una costante computazionale di tipo *value* deve essere dato da L.

- $* : 1$. L'interpretazione del tipo 1 avviene nell'oggetto terminale $1 = \{*\}$ di **CPO**. L'unica possibilità in questo caso è quella di porre

  $$[\![*]\!] = * \in 1 \tag{5.8.8}$$

- $read_x : T_E \ value$. Una configurazione di memoria $s \in S$ è data da una funzione $\mathcal{V} \to N$. Inoltre, si è assunto di avere a disposizione nella categoria un oggetto $\mathcal{V} \to Q$ i cui elementi costituiscono ognuno una descrizione dei livelli di sicurezza delle variabili (spesso, in letteratura, tali elementi vengono detti **tipaggi**, in analogia con quanto avviene nell'analisi statica di identificare gli elementi di Q quali tipi delle variabili). Si effettua l'assunzione che il tipaggio sia fissato e che corrisponda ad un elemento $\tau \in \mathcal{Q}^{\mathcal{V}}$. Da ciò ne consegue che l'interpretazione viene definita per un solo tipaggio τ.
  L'interpretazione della primitiva computazionale $read_x$ in una configurazione di memoria di s è data dalla coppia contenente il valore della variabile (rispetto alla configurazione di memoria stessa), e il livello di sicurezza che il tipaggio τ associa a tale variabile.
  $$[\![read_x]\!]s =< s(x),\tau(x) >\in N \times Q \tag{5.8.9}$$

- $(m : value,n : value) \Rightarrow \underline{\theta}(m,n) : value$. L'interpretazione di un operatore θ non è definibile in maniera generica per tutti gli operatori. Occorre difatti effettuare una distinzione su come tale operatore agisce sul livello di sicurezza di un'espressione. Si consideri, ad esempio, un espressione del tipo

  $$\text{h} + 5$$

  Ci si chiede quale sia il livello di sicurezza associato a tale espressione. Il livello di sicurezza della costante 5 viene definito per essere L, mentre per la variabile h si ottiene che essa ha livello di sicurezza pari a $\tau(h) = H$. In questo caso, è ragionevole porre il livello di sicurezza dell'intera espressione ad H, in quanto il valore di questa dipende da delle variabili alte. In generale, si può dire che il livello di sicurezza associato ad $E_1 + E_2$ è dato dal massimo tra i livelli di sicurezza delle singole espressioni.
  Tuttavia, ciò è vero per l'operatore + in quanto esso è un **operatore statico**, ovvero in ogni caso verranno valutate entrambi gli operandi. Tuttavia, esistono dei casi di operatori **dinamici**, in cui la valutazione del secondo operando dipende strettamente dalla valutazione del precedente (o viceversa). Si consideri, ad esempio, l'operatore || di or, definito all'interno di molti linguaggi di programmazione quali **C** e **Java**.
  Si consideri, dunque, un'espressione del tipo

  $$\text{l} \;||\; \text{h}$$

  In questo caso, se la valutazione di l risulta diversa da 0, non si procederà a valutare la variabile h, e il livello di sicurezza dell'intera espressione sarà dato da L. In caso contrario, si procederà a valutare h, associando all'intera espressione il valore H.
  Per ogni operatore $\underline{\theta}$, si assume di disporre di una funzione $level_\theta : (nat \times Q) \times (nat \times Q) \to Q$ che definisce il livello di sicurezza di un'espressione a partire dai valori e dai livelli di sicurezza delle sue sottoespressioni. Nel caso in cui l'operatore θ sia statico, si pone
  $$level_\theta(n_1,q_1,n_2,q_2) = \max\{q_1,q_2\}$$

Per un generico operatore $\underline{\theta}$ si può esprimere la sua interpretazione come

$$[\![\underline{\theta}(m,n)]\!] =< \pi_1[\![m]\!]\theta\pi_1[\![n]\!], level_\theta(m,n) > \tag{5.8.10}$$

- $n : value \Rightarrow upd_x\ n : T_C\ 1$. L'operatore di aggiornamento rappresenta il cuore della semantica per la rilevazione dei flussi di informazione. E' in questo caso, infatti, che si stabilisce se è avvenuto un flusso di informazioni dalla memoria alta alla memoria bassa.
  L'esito di un update dipende, dal punto di vista della sicurezza, dal seguente insieme di parametri:
  - Dal contesto di sicurezza che viene utilizzato per l'interpretazione della primitiva. Ad esempio, si prenda in considerazione il caso in cui avviene un aggiornamento di una variabile bassa in un contesto di sicurezza alto. In questo caso, è avvenuto un flusso di informazioni, in quanto l'utilizzo del contesto di sicurezza H indica che il risultato dell'intepretazione di tale comando dipende da una variabile alta; il risultato dell'interpretazione dell'aggiornamento dovrà quindi corrispondere con lo stato di errore $* \in 1$.
  - Dal livello di sicurezza che viene passato all'operatore upd_x. Difatti, anche assumendo che l'aggiornamento avvenga in un contesto di sicurezza basso, se il valore che viene passato all'aggiornamento di una variabile bassa ha un livello di sicurezza pari ad H, ancora una volta l'esito dell'aggiornamento corrisponderà ad una configurazione di errore.
  - Nei restanti casi, ovvero, in cui si ha l'aggiornamento di una variabile alta oppure un aggiornamento di una variabile bassa in un contesto di sicurezza basso e con un valore avente livello di sicurezza basso, si dovrà procedere ad aggiornare il valore della variabile in base alla configurazione di memoria passata in input.

  Nel linguaggio matematico, verranno utilizzate le notazioni $[\![\cdot]\!]_L$ e $[\![\cdot]\!]_H$ per indicare le interpretazioni nei rispettivi contesti di sicurezza. Si ottiene

$$\begin{aligned}
[\![upd_x\ n]\!]_L s \quad = \quad & case\ \tau(x)\ of \\
& L.case\ \pi_2[\![n]\!]\ of \\
& \quad L.val_\perp\ \iota_1(< *, s[x \leftarrow \pi_1[\![n]\!]] >) \\
& \quad H.val_\perp\ \iota_2(*) \\
& H.val_\perp\ \iota_1(< *, s[x \leftarrow \pi_1[\![n]\!]] >)
\end{aligned} \tag{5.8.11}$$

  Se, invece, il contesto di sicurezza in cui avviene l'interpretazione è dato da H, l'interpretazione dell'aggiornamento è definita come

$$\begin{aligned}
[\![upd_x\ n]\!]_H s \quad = \quad & case\ \tau(x)\ of \\
& L.val_\perp\ \iota_2(*) \\
& H.val_\perp\ \iota_1(< *, s[x \leftarrow \pi_1[\![n]\!]] >)
\end{aligned} \tag{5.8.12}$$

- $n : value, M : T_C\ 1, N : T_C\ 1 \Rightarrow cond(n,M,N) : T_C\ 1$. In questo caso, si dovrà procedere a selezionare, oltre al termine da interpretare in base al valore in cui viene valutato n, anche il contesto di sicurezza in cui tale termine dovrà essere interpretato. Distinguiamo due casi:

  – L'intero termine viene interpretato in un contesto di sicurezza alto. In questo caso, indipendentemente dal livello di sicurezza del valore n, si procederà a selezionare il termine da valutare (in base al valore intero contenuto in n) nel contesto alto.

  – L'intero termine viene interpretato in un contesto di sicurezza basso. Differentemente dal caso precedente, il livello di sicurezza del valore n determina il contesto di sicurezza in cui interpretare il termine selezionato.

  Formalmente:

$$\begin{aligned} [\![cond(n,M,N)]\!]_L s &= case\ \pi_1[\![n]\!]\ of \\ &\quad 0.[\![N]\!]_{\pi_2[\![n]\!]} s \\ &\quad s(n).[\![M]\!]_{\{\pi_2[\![n]\!]\}} s \qquad (5.8.13) \\ [\![cond(n,M,N)]\!]_H s &= case\ \pi_1[\![n]\!]\ of \\ &\quad 0.[\![N]\!]_H s \\ &\quad s(n).[\![M]\!]_H s \qquad (5.8.14) \end{aligned}$$

- $\Gamma, x : A \vdash M : B \Rightarrow \emptyset \vdash (fn\ x : A \Rightarrow M) : A \to B$, $M : A \to B, N : A \Rightarrow MN : B$. In questo caso, in quanto non si devono considerare gli effetti computazionali introdotti, l'interpretazione rimane identica a quella definita in 4.5. Per completezza, vengono riportate entrambe le equazioni:

$$[\![fn\ x : A \Rightarrow M]\!] = \lambda x : [\![A]\!].[\![M]\!]_{x:A} \qquad (5.8.15)$$

$$[\![MN]\!] : [\![M]\!]([\![N]\!]) \qquad (5.8.16)$$

- $(M : T_C\ 1 \to T_C\ 1) \Rightarrow fix\ M : T_C\ 1$. Poiché l'oggetto $[\![T_C\ 1]\!]$ è dato da un dominio, è possibile applicare un operatore di punto fisso per interpretare tale termine. Si ottiene:

$$[\![fix\ M]\!] = \mu x.[\![M]\!]_{x:T_C\ 1} \qquad (5.8.17)$$

- La definizione degli operatori let_C, val_C, let_E e val_E è stata data in 3. Per semplicità, tali equazioni vengono omesse, e si rimanda il lettore a 3 per ulteriori dettagli.

- Infine, si deve definire l'interpretazione dell'operatore di conversione ε introdotto in 5.6.1. Sia $M : T_E\ A$. Tale termine può essere convertito in un termine di tipo $T_C\ A$ ignorando il contesto di sicurezza che viene passato in input, e restituendo la configurazione di memoria utilizzata come argomento. Si ha

$$[\![\varepsilon M]\!] = \lambda \overline{q} : Q.\lambda s : S.val_\perp\ \iota_1(< [\![M]\!]s, s >) \qquad (5.8.18)$$

5.9 Proprietà dell'interpretazione

In questo capitolo sono stati analizzati gli aspetti della sicurezza di un linguaggio di programmazione, relativamente ai flussi di informazione. Nel corso della discussione si è definita una semantica che permetta di analizzare i programmi e di rilevare tali flussi tra la memoria alta e la memoria bassa.
Per concludere la trattazione del problema, si vuole mostrare che gli strumenti qui sviluppati modellano correttamente il problema preso in analisi, facendo coincidere l'interpretazione di programmi interferenti (utilizzando un contesto di sicurezza basso, e da una memoria opportunamente selezionata) con lo stato di errore introdotto.
Ad ogni modo, si mette in evidenza che la soluzione proposta non descrive **completamente** il problema dell'information flow: si esibiranno, in seguito, esempi di comandi non interferenti per i quali l'interpretazione $[\![(\!(C)\!)]\!]_L$ è data dallo stato di errore.

5.9.1 Interpretazione dei comandi

I risultati che verranno esposti riguardano la sola interpretazione dei comandi di **TinyC**. Per semplificare l'esposizione di tali teoremi, verrà data l'interpretazione per intero dei comandi, ottenuta applicando ad essi prima la traduzione definita in 5.7, quindi interpretando i termini del metalinguaggio ottenuti secondo quanto definito in 5.8. Per brevità, non si esibiranno i calcoli attraverso i quali si ottiene l'interpretazione dei termini. Inoltre, per semplicità, si assumerà che non vengano utilizzati operatori dinamici all'interno delle espressioni.
Per le espressioni di **TinyC** si ottiene

$$[\![\mathrm{n}]\!]s = < n,L > \tag{5.9.1}$$
$$[\![\mathrm{x}]\!]s = < s(x),\tau(x) > \tag{5.9.2}$$
$$[\![\mathrm{E}_1\ \theta\ \mathrm{E}_2]\!]s = < \pi_1([\![\mathrm{E}_1]\!]s)\theta\pi_2([\![\mathrm{E}_2]\!]s), \max\{\pi_2([\![\mathrm{E}_1]\!]s),\pi_2([\![\mathrm{E}_1]\!]s)\} > \tag{5.9.3}$$

Dove si ricorda che $[\![\mathrm{E}]\!]s \in N \times Q$. Dunque, $\pi_1([\![\mathrm{E}]\!]s)$ corrisponde al valore intero risultato dell'espressione, mentre $\pi_2([\![\mathrm{E}_1]\!]s)$ rappresenta il livello di sicurezza associato all'espressione.
Per ogni comando dovranno essere definite le interpretazioni nei contesti di sicurezza alto e basso: quando l'interpretazione di un comando non dipenderà dal contesto di sicurezza

in cui tale comando viene interpretato, si userà la notazione $[\![\cdot]\!]_q$

$$[\![\text{skip}]\!]_q s = val_\perp\ \iota_1 < *,s > \tag{5.9.4}$$

$$\begin{aligned}[\![\text{x:=E}]\!]_L s = {} & val_\perp(case\ \tau(x)\ of \\ & L.case\ \pi_2([\![\text{E}]\!]s)\ of \\ & \quad L.\iota_1(< *,s[x \leftarrow \pi_1([\![\text{E}]\!]s)] >) \\ & \quad H.\iota_2(*) \\ & H.\iota_1(< *,s[x \leftarrow \pi_1([\![\text{E}]\!]s)] >))\end{aligned} \tag{5.9.5}$$

$$\begin{aligned}[\![\text{x := E}]\!]_H s = {} & val_\perp(case\ \tau(x)\ of \\ & L.\iota_2(*) \\ & H.\iota_1(< *,s[x \leftarrow \pi_1[\![\text{E}]\!]s] >))\end{aligned} \tag{5.9.6}$$

$$\begin{aligned}[\![\text{C}_1;\text{C}_2]\!]_q s = {} & let_\perp\ x = [\![\text{C}_1]\!]_q\ in \\ & case\ x\ of \\ & \quad \iota_1(< \overline{x},s' >).[\![\text{C}_2]\!]_q s' \\ & \quad \iota_2(*).val_\perp\ \iota_2(*) >\end{aligned} \tag{5.9.7}$$

$$\begin{aligned}[\![\text{if E then C}_1\text{ else C}_2]\!]_L s = {} & case\ \pi_1([\![\text{E}]\!]s)\ of \\ & 0.[\![\text{C}_2]\!]_{\pi_2([\![\text{E}]\!]s)}\ s \\ & succ(n).[\![\text{C}_1]\!]_{\pi_2([\![\text{E}]\!]s)}\ s\end{aligned} \tag{5.9.8}$$

$$\begin{aligned}[\![\text{if E then C}_1\text{ else C}_2]\!]_H s = {} & case\ \pi_1([\![\text{E}]\!]s)\ of \\ & 0.[\![\text{C}_2]\!]_H\ s \\ & succ(n).[\![\text{C}_1]\!]_H\ s\end{aligned} \tag{5.9.9}$$

(5.9.10)

$$
\begin{aligned}
[\![\text{while E do C}]\!]_L s \quad = \quad & (\mu f.\lambda s : S.case\ \pi_1([\![\text{E}]\!]s)\ of \\
& 0.val_\perp\ \iota_1(<*,s>) \\
& succ(n).let_\perp\ x = [\![\text{C}]\!]_{\pi_2([\![\text{E}]\!]s)}\ in \\
& \quad case\ x\ of \\
& \quad \iota_1(\overline{x},s').f_{\pi_2([\![\text{E}]\!]s)}s' \\
& \quad \iota_2(*).val_\perp\ \iota_2(*))s
\end{aligned} \tag{5.9.11}
$$

$$
\begin{aligned}
[\![\text{while E do C}]\!]_H s \quad = \quad & (\mu f.\lambda s : S.case\ \pi_1([\![\text{E}]\!]s)\ of \\
& 0.val_\perp\ \iota_1(<*,s>) \\
& succ(n).let_\perp\ x = [\![\text{C}]\!]_H\ in \\
& \quad case\ x\ of \\
& \quad \iota_1(\overline{x},s').f_H s' \\
& \quad \iota_2(*).val_\perp\ \iota_2(*))s
\end{aligned} \tag{5.9.12}
$$

dove f_q denota che al parametro f deve essere applicato il contesto di sicurezza q.

5.9.2 Correttezza

Il primo risultato che si vuole dimostrare è dato dalla correttezza dell'interpretazione dei comandi. Tale risultato mette in relazione l'interpretazione definita per i comandi di TinyC in questa sezione con quella definita in 4.6. In termini descrittivi, qualora la computazione associata ad un comando non risulti nello stato di errore $val_\perp\ (\iota_2\ (*))$, allora la configurazione di memoria calcolata coincide con quella dell'interpretazione 4.6 (o, nel caso in cui si ottenga $\perp$, tale risultato è identico in entrambe le interpretazioni).
Prima di introdurre formalmente il teorema, si dovrà dimostrare un risultato analogo per le espressioni. Per rendere chiaro a quali delle due semantiche si stia facendo riferimento durante la dimostrazione dei risultati, nel seguito l'interpretazione esibita in 4.6 verrà denotata con $\mathcal{S}[\![\cdot]\!]$.

Lemma 5.9.1. *Sia* $[\![E]\!]s =< n,q >$. *Allora* $n = v(E,s)$.

Dimostrazione. Per induzione sulla struttura delle espressioni. □

Teorema 5.9.2 (Correttezza)**.** *Sia* $[\![C]\!]_q s \neq val_\perp \iota_2(*)$. *Allora vale la seguente relazione:*

$$
\begin{aligned}
[\![C]\!]_q s = \perp \quad &\Rightarrow \quad \mathcal{S}[\![C]\!]s = \perp \\
[\![C]\!]_q s = val_\perp\ \iota_1(<*,s'>) \quad &\Rightarrow \quad \mathcal{S}[\![C]\!]s = val_\perp\ <*,s'>
\end{aligned}
$$

Dimostrazione. Per induzione sulla struttura dei comandi.

- C ≡ skip. Questo caso è banale, in quanto

$$\begin{aligned} [\![\text{skip}]\!]_q\ s &= val_\perp \iota_1 < *,s > \\ \mathcal{S}[\![\text{skip}]\!]s &= val_\perp < *,s > \end{aligned}$$

- C ≡ x := E. Si considererà il solo caso di $[\![\text{x:=E}]\!]_H s$, per il contesto L si può procedere analogamente.
  Dall'equazione 5.9.6 si deve avere forzatamente $\tau(x) = H$, in quanto si è assunto per ipotesi che $[\![\text{x := E}]\!]_H s \neq val_\perp(\iota_2(*))$. In questo caso l'equazione può essere ridotta a:

$$[\![\text{x := E}]\!]_H s = val_\perp \iota_1(< *,s[x \leftarrow \pi_1([\![\text{E}]\!]s)] >)$$

  Dal lemma 5.9.1 si può sostituire $\pi_1([\![\text{E}]\!]s)$ con $v(\text{E},s)$. Poiché

$$\mathcal{S}[\![\text{x := E}]\!]s = val_\perp < *,s[x \leftarrow v(\text{E},s)] >$$

  la relazione risulta valida in questo caso.

- C ≡ C_1 ; C_2. Dall'equazione 5.9.7, e dall'ipotesi effettuata, si deve avere necessariamente $[\![\text{C}_1]\!]_q s \neq val_\perp \iota_2(*)$. Se così fosse, difatti, si otterrebbe $[\![\text{C}_1\ ;\ \text{C}_2]\!]_q s = val_\perp \iota_2(*)$, contro l'assunzione effettuata. In questo caso, vi sono due casi possibili.
  - $[\![\text{C}_1]\!]_q s = \perp$. Si avrebbe dunque $[\![\text{C}_1\ \text{C}_2]\!] = \perp$. Per ipotesi induttiva, inoltre, si otterrebbe $\mathcal{S}[\![\text{C}_1]\!] = \perp$, e dunque $\mathcal{S}[\![\text{C}_1\ ;\ \text{C}_2 = \perp]\!]$.
  - $[\![\text{C}_1]\!]_q s = val_\perp\ \iota_1(< *,s' >)$. In questo caso, si otterrebbe $[\![\text{C}_1\ ;\ \text{C}_2]\!]_q s = [\![\text{C}_2]\!]s'$. Per ipotesi induttiva, si ottiene inoltre $\mathcal{S}[\![\text{C}_1]\!]s = val_\perp\ \ < *,s' >$, e dunque $\mathcal{S}[\![\text{C}_1\ ;\ \text{C}_2]\!]s' = \mathcal{S}[\![\text{C}_2]\!]s'$. La validità della relazione si ottiene applicando nuovamente l'ipotesi induttiva a C_2.

- C ≡ if E then C_1 else C_2: si considererà il solo caso di applicazione al contesto di sicurezza H. Applicando il lemma 5.9.1 all'equazione 5.9.9, quest'ultima si riscrive come

$$\begin{aligned} [\![\text{if E then } \text{C}_1 \text{ else } \text{C}_2]\!]_H s &= case\ v(\text{E},s)\ of \\ &\quad\ z.[\![\text{C}_2]\!]_H s \\ &\quad\ succ(n).[\![\text{C}_1]\!]_H s \end{aligned}$$

  Senza perdita di generalità, sia $v(\text{E},s) = 0$. Si ottengono le seguenti equazioni:

$$\begin{aligned} [\![\text{if E then } \text{C}_1 \text{ else } \text{C}_2]\!]_H s &= [\![\text{C}_2]\!]_H s \\ \mathcal{S}[\![\text{if E then } \text{C}_1 \text{ else } \text{C}_2]\!]s &= \mathcal{S}[\![\text{C}_2]\!]s \end{aligned}$$

  Il risultato segue dall'ipotesi induttiva applicata a C_2. Il caso in cui il contesto di sicurezza applicato è L viene gestito in maniera analoga.

- C ≡ while E do C_1. Anche in quest'ultimo caso, si considererà il solo caso del contesto di sicurezza H. Applicando il lemma 5.9.1 all'equazione 5.9.12, essa viene riscritta come:

$$\begin{aligned}[\![\text{while E do } C_1]\!]_H s \;=\; & (\mu f.\lambda s : S.case\ v(\text{E},s)\ of \\ & z.val_\perp \iota_1(<*,s>) \\ & succ(n).let_\perp\ x = [\![C_1]\!]_H s\ in \\ & \ case\ x\ of \\ & \ \iota_1(<*,s'>).f_H s' \\ & \ \iota_2(*).val_\perp\ \iota_2(*))s\end{aligned}$$

Si costruisca la successione di elementi di tipo $((1 \times S) + 1)_\perp$ come segue:

$$\begin{aligned}d_0 \;=\; & case\ \pi_1([\![E]\!]s);\ of \\ & z.val_\perp\ \iota_1(<*,s>) \\ & succ(n).[\![C]\!]_H s\end{aligned}$$

$$\begin{aligned}d_n \;=\; & let_\perp\ x = d_{n-1}\ in \\ & case\ x\ of \\ & \iota_1(<*,s>).case\ \pi_1([\![E]\!])s\ of \\ & \ z.val_\perp\ \iota_1(s) \\ & \ succ(n).[\![C]\!]_H\ s \\ & \iota_2(*).val_\perp \iota_2(*)\end{aligned}$$

Nella successione d_0,$d_1, \cdots$ l'elemento d_i consiste nel valore ottenuto all'esecuzione dell'i-esima iterazione del ciclo while (oppure consiste nello stato di errore, nel caso in cui venga ottenuto uno stato di errore in un'iterazione precedente del ciclo). Allo stesso modo si costruisca la successione di elementi di tipo $(1 \times S)_\perp$ come segue:

$$\begin{aligned}c_0 \;=\; & let_\perp\ x = \mathcal{S}[\![E]\!]s\ \ in \\ & \ case\ x\ of \\ & \ z.val_\perp\ <*,s> \\ & \ succ(n).\mathcal{S}[\![C]\!]s\end{aligned}$$

$$\begin{aligned}c_n \;=\; & let_\perp\ <*,s> = c_{n-1}\ in \\ & \ let_\perp\ x = \mathcal{S}[\![E]\!]s\ \ in \\ & \ case\ x\ of \\ & \ z.val_\perp\ <*,s> \\ & \ succ(n).\mathcal{S}[\![C]\!]s\end{aligned}$$

Dall'ipotesi induttiva e dal lemma 5.9.1 si ottiene $\forall i.d_i \neq val_\perp\ \iota_2(*)$:

$$\begin{aligned} d_i = \perp &\Rightarrow c_i = \perp \\ d_i = val_\perp\ \iota_1(<*,s>) &\Rightarrow c_i = val_\perp\ <*,s> \end{aligned}$$

Per ipotesi si suppone che $[\![\text{while E do } C_1]\!]_H s \neq val_\perp\ \iota_2(*)$: in questo caso si deve avere $\forall i.d_i \neq val_\perp\ \iota_2(*)$. Se infatti esistesse un indice i per il quale $d_i = val_\perp\ \iota_2(*)$ si avrebbe che l'i-esima iterazione del ciclo while produce come risultato della computazione lo stato di errore, e dunque l'intero comando verrebbe valutato in tale stato. Si ottiene dunque che i valori d_i e c_i ottenuti ad ogni iterazione del ciclo while soddisfano la proprietà di correttezza che si vuole dimostrare.

Si supponga dunque $[\![\text{while E do } C_1]\!]s = val_\perp\ \iota_1(<*,s'>)$, e sia n il numero di iterazioni effettuate dal ciclo while.
Il valore della successione costruita d_n è dato esattamente da $val_\perp\ \iota_1(<*,s'>)$, e dunque $c_n = val_\perp\ <*,s'>$. Dal lemma 5.9.1 si ottiene inoltre che $c_n = \mathcal{S}[\![\text{while E do } C_1]\!]s$, e dunque in questo caso non vi è nulla da dimostrare.

Differentemente, sia $[\![\text{while E do } C_1]\!]s = \perp$. Vi sono due casi possibili:

- $\exists i.d_i = \perp$: in questo caso $c_i = \perp$, e dunque $\mathcal{S}[\![whileE]\!]s = \perp$.

- Nel secondo caso nessuno degli elementi della successione $d_0,d_1,\cdots$ valuta in $\perp$: dunque si può porre $d_i = val_\perp\ \iota_1(<*,s_0>)$. Affinché sia $[\![\text{while E do } C_1]\!]s = \perp$, si deve avere $\forall i.v(E,s_i) \neq 0$. Dal lemma 5.9.1 si ottiene che $\mathcal{S}[\![\text{while E do } C_1]\!]s = \perp$.

□

5.9.3 Soundness

La proprietà principale di cui l'interpretazione esibita gode può essere sintetizzata nel fatto che l'interferenza di un comando corrisponde all'interpretazione di tale comando (nel contesto di sicurezza L e in un'opportuna memoria s) nello stato di errore $val_\perp\iota_2(*)$. Prima di procedere con la dimostrazione del risultato, si illustrerà che l'interpretazione di un comando in uno stato di errore non coincide esattamente con l'interferenza di tale comando, id est esistono dei comandi non interferenti che vengono interpretati nello stato di errore. Si consideri, difatti, il seguente comando C:

```
1 l := h;
2 l := 0;
```

E' immediato notare che tale comando è non interferente: tuttavia, si scelga una memoria arbitraria s. Dalle equazioni 5.9.5 e 5.9.7 si può calcolare l'interpretazione del comando

sopra citato:

$$\begin{aligned}
[\![\mathrm{C}]\!]_L s &= let_\perp\ x = [\![\mathrm{l} := \mathrm{h}]\!]_L s\ in \\
&\quad case\ x\ of \\
&\quad\ \iota_1(<\overline{x}, s'>).[\![\mathrm{l}{:=}0]\!]_L s' \\
&\quad\ \iota_2(*).val_\perp\ \iota_2(*)
\end{aligned}$$

Per $[\![\mathrm{l}{:=}\mathrm{h}]\!]_L s$ si ottiene

$$\begin{aligned}
[\![l := h]\!]_L s &= val_\perp(case\ \tau(l)\ of \\
&\quad L.case\pi_2([\![h]\!]\ s)\ of \\
&\quad\ L.\iota_1(<*, s[l \leftarrow \pi_1([\![h]\!]s)]>) \\
&\quad\ H.\iota_2(*) \\
&\quad H.\iota_1(<*, s[l \leftarrow \pi_1([\![h]\!]s)]>)) \\
&= val_\perp\ \iota_2(*)
\end{aligned}$$

Andando a sostituire, nell'interpretazione di C, si ricava la soluzione dell'equazione:

$$\begin{aligned}
[\![\mathrm{C}]\!]_L s &= let_\perp\ x = val_\perp\ \iota_2(*)\ in \\
&\quad case\ x\ of \\
&\quad\ \iota_1(<\overline{x}, s'>).[\![\mathrm{l}{:=}0]\!]_L s' \\
&\quad\ \iota_2(*).val_\perp\ \iota_2(*) \\
&= case\ \iota_2(*)\ of \\
&\quad\ \iota_1(<\overline{x}, s'>).[\![\mathrm{l}{:=}0]\!]_L s' \\
&\quad\ \iota_2(*).val_\perp\ \iota_2(*) \\
&= val_\perp\ \iota_2(*)
\end{aligned}$$

Si è dunque mostrato che esistono degli esempi di comandi non interferenti che vengono interpretati nello stato di errore.
Sebbene l'esempio dato illustri che il concetto di non interferenza non coincide con l'intepretazione dei comandi nello stato di errore, è possibile comunque provare che la semantica definita è **Sound**, ovvero

Teorema 5.9.3 (Soundness)**.** *Sia C interferente. Per definizione esistono due memorie* $s_1 =_L s_2$ *tali che* $\mathcal{S}[\![C]\!]s_1 \not=_L \mathcal{S}[\![C]\!]s_2$*. Allora vale almeno una delle seguenti:*

- $[\![C]\!]s_1 = val_\perp\ \iota_2(*)$
- $[\![C]\!]s_2 = val_\perp\ \iota_2(*)$

Prima di esibire la prova del teorema è necessario introdurre dei lemmi che verranno applicati durante lo svolgimento della dimostrazione:

Lemma 5.9.4. $[\![C]\!]_L s = val_\perp\ \iota_2(*) \Rightarrow [\![C]\!]_H s = val_\perp\ \iota_2(*)$

Dimostrazione. E' sufficiente notare che l'unica primitiva che può generare lo stato di errore è data dall'assegnamento. Nel caso in cui un assegnamento del tipo x:=E venga interpretato in un contesto di sicurezza L, si deve avere necessariamente $\pi_2([\![E]\!]s) = H$, $\tau(x) = L$. Differentemente, nel caso in cui a tale espressione venga applicato il contesto di sicurezza H, è sufficiente che venga effettuato un assegnamento di variabile bassa.
Il lettore interessato ai dettagli può procedere per induzione sulla struttura dei comandi. Per il caso del while, supponendo che $[\![C]\!]_L = val_\perp\ \iota_2(*)$ (ovvero non viene generata non terminazione), è sufficiente procedere per induzione sul numero di iterazioni effettuate dal comando while. □

Lemma 5.9.5. *Se in $[\![C]\!]_H s$ viene valutato un assegnamento di variabile bassa, allora $[\![C]\!]_H s = val_\perp\ \iota_2(*)$.*

Dimostrazione. Anche in questo caso è sufficiente notare che l'assegnamento di variabile bassa è l'unico caso in cui è possibile ottenere lo stato di errore nel contesto di sicurezza H. Dunque, se avviene un assegnamento di variabile bassa, allora l'intero comando valuterà a $val_\perp\ \iota_2(*)$. □

Dimostrazione 5.9.3. Per induzione sulla struttura dei termini.

- C ≡ skip. Il comando skip non è di per sè interferente: dunque, non vi è niente da dimostrare.
- C ≡ x :=E. Per ipotesi, $\exists s_1 =_L s_2 . \mathcal{S}[\![x := E]\!]s_1 \not=_L \mathcal{S}[\![x := E]\!]s_2$. Affinché ciò sia vero, si deve avere necessariamente che in E appaia una variabile alta. In caso contrario, si avrebbe

$$\begin{aligned} \mathcal{S}[\![\text{x := E}]\!]s_1 &= val_\perp\ < *, s_1[x \leftarrow v(\text{E}, s_1)] > \\ \mathcal{S}[\![\text{x := E}]\!]s_2 &= val_\perp\ < *, s_2[x \leftarrow v(\text{E}, s_2)] > \end{aligned}$$

  Essendo $s_1 =_L s_2$, si ha $v(\text{E},s_1) = v(\text{E},s_2)$.
  Inoltre, affinché sia avvenuto un flusso di informazioni deve valere $\tau(x) = L$. Dalle equazioni sopra esibite, se così non fosse si avrebbe

$$s_1[x \leftarrow v(\text{E},s_1)] =_L s_1 =_L s_2 =_L s_2[x \leftarrow v(\text{E},s_2)]$$

  E' immediato notare che, se una variabile alta appare in E, allora $\forall s . \pi_2([\![\text{E}]\!]s) = H$ (si ricorda che, per semplicità, si sta considerando il caso in cui gli operatori θ siano statici). L'equazione 5.9.5 viene ridotta in questo caso a

$$[\![\text{x:=E}]\!]s = val_\perp\ \iota_2(*)$$

  Dove s rappresenta una memoria arbitraria.
- C ≡ C_1 ; C_2. Vi sono due casi da distinguere:

- il flusso di informazioni avviene in C_1. In questo caso, per ipotesi induttiva, vale $[\![\mathrm{C}_1]\!]s_1 = val_\perp\ \iota_2(*)$ opppure $[\![\mathrm{C}_1]\!]s_2 = val_\perp\ \iota_2(*)$. Senza perdita di generalità, si assuma che il comando C_1 venga interpretato nello stato di errore quando ad esso viene applicato la memoria s_1. Applicando l'equazione 5.9.7 si ottiene $[\![\mathrm{C}_1\ ;\ \mathrm{C}_2]\!]s_1 = val_\perp\ \iota_2(*)$.
- $[\![\mathrm{C}_1$ non provoca flusso di informazioni. Si è mostrato che esistono casi di comandi non interferenti che vengono valutati nello stato di errore: se questo è il caso, ci si può ricondurre a quanto appena discusso. Sia pertanto $[\![\mathrm{C}_1]\!]s_1 = val_\perp\ \iota_1(<*,s_1'>)$, $[\![\mathrm{C}_2]\!]s_2 = val_\perp\ \iota_1(<*,s_2'>)$. Dall'ipotesi induttiva si ha $s_1 =_L s_2$, e dal teorema di correttezza,

$$\begin{aligned}\mathcal{S}[\![\mathrm{C}_1]\!]s_1 &= val_\perp <*,s_1'>\\ \mathcal{S}[\![\mathrm{C}_1]\!]s_2 &= val_\perp <*,s_2'>\end{aligned}$$

  Poiché si è supposto che $\mathrm{C}_1\ ;\ \mathrm{C}_2$ è interferente, deve valere

$$\mathcal{S}[\![\mathrm{C}_2]\!]s_1' \not\approx \mathcal{S}[\![\mathrm{C}_2]\!]s_2'$$

  Dunque, dall'ipotesi induttiva applicata a C_2, vale una delle seguenti:

$$\begin{aligned}[\![\mathrm{C}_2]\!]s_1' &= val_\perp\ \iota_2(*)\\ [\![\mathrm{C}_2]\!]s_2' &= val_\perp\ \iota_2(*)\end{aligned}$$

  Senza perdita di generalità, sia s_1' la memoria che genera lo stato di errore: sostituendo nell'equazione 5.9.7 si ottiene

$$[\![\mathrm{C}_1\ ;\ \mathrm{C}_2]\!] = val_\perp \iota_2(*)$$

- $\mathrm{C} \equiv$ if E then C_1 else C_2: Sia $s_1 =_L s_2$, e sia

$$\mathcal{S}[\![\text{if E then } \mathrm{C}_1 \text{ else } \mathrm{C}_2]\!]s_1 \not\approx_L \mathcal{S}[\![\text{if E then } \mathrm{C}_1 \text{ else } \mathrm{C}_2]\!]s_2$$

  Si devono distinguere due casi:

  - In questo caso in E deve necessariamente comparire una variabile alta, e dunque $\forall s.\pi_2([\![E]\!]s) = H$. In questo caso, devono essere distinti diversi casi in base ai valori di $v(\mathrm{E},s_1)$ e $v(\mathrm{E},s_2)$.
    Nel caso in cui $v(E,s_1) \neq 0$, $v(E,s_2) \neq 0$, si ottiene

$$\begin{aligned}[\![\text{if E then } \mathrm{C}_1 \text{ else } \mathrm{C}_2]\!]s_1 &= [\![\mathrm{C}_1]\!]_H s_1\\ [\![\text{if E then } \mathrm{C}_1 \text{ else } \mathrm{C}_2]\!]s_2 &= [\![\mathrm{C}_1]\!]_H s_2\end{aligned}$$

    Dall'ipotesi induttiva si ha che $[\![\mathrm{C}_1]\!]_L s_1 = val_\perp\ \iota_2(*)$, oppure $[\![\mathrm{C}_2]\!]_L s_2 = val_\perp\ \iota_2(*)$. Applicando il lemma 5.9.4, si ottiene la validità del teorema.
    Si consideri invece il caso in cui $v(\mathrm{E},s_1) \neq 0$, $v(\mathrm{E},s_2) = 0$. Si ha

$$\begin{aligned}\mathcal{S}[\![\text{if E then } \mathrm{C}_1 \text{ else } \mathrm{C}_2]\!]s_1 &= \mathcal{S}[\![\mathrm{C}_1]\!]s_1\\ \mathcal{S}[\![\text{if E then } \mathrm{C}_1 \text{ else } \mathrm{C}_2]\!]s_2 &= \mathcal{S}[\![\mathrm{C}_2]\!]s_2\end{aligned}$$

Per ipotesi vale $\mathcal{S}[\![\mathrm{C}_1]\!]s_1 \neq_L \mathcal{S}[\![\mathrm{C}_2]\!]s_2$. Affinché ciò sia vero, in almeno una di queste due computazioni deve essere stato effettuato un assegnamento di memoria bassa. Il risultato segue dunque dal lemma 5.9.5.

– $v[\mathrm{E},s_1] = v[\mathrm{E},s_2]$. Ciò è possibile sia se $\pi_2([\![E]\!]s) = H$, sia se $\pi_2([\![E]\!])s = L$, dove s è uno stato arbitrario (dal momento che si assume di utilizzare solamente operatori statici, questi sono gli unici due casi possibili). Dal lemma 5.9.4, è sufficiente dimostrare il solo caso in cui $\pi_2([\![E]\!]s) = L$.
Senza perdita di generalità, sia $v(\mathrm{E},s_1) \neq 0$: si ha

$$\begin{aligned} [\![\text{if E then } \mathrm{C}_1 \text{ else } \mathrm{C}_2]\!]_L s_1 &= [\![\mathrm{C}_1]\!]_L s_1 \\ [\![\text{if E then } \mathrm{C}_1 \text{ else } \mathrm{C}_2]\!]_L s_2 &= [\![\mathrm{C}_1]\!]_L s_2 \end{aligned}$$

Per ipotesi, deve essere $\mathcal{S}[\![\mathrm{C}_1]\!]s_1 \neq_L \mathcal{S}[\![\mathrm{C}_2]\!]$, e il risultato segue dall'ipotesi induttiva.

- C $\equiv$ while E do C_1. Siano $s_1 =_L s_2$ per le quali il comando è interferente. Dalla definizione 5.1.2, si deve necessariamente avere che in entrambi i casi il comando termini in uno stato di memoria (e dunque non è possibile il caso della non terminazione). Siano dunque n ed m rispettivamente il numero di iterazioni effettuate nel caso di s_1 ed s_2. Si procede per induzione su n.

  $n = 0$**:** Ciò avviene solamente se $v(E,s_1) = 0$. Affinché si abbia interferenza, deve inoltre essere $v(E,s_2) \neq 0$: in caso contrario, difatti, ci si ricondurrebbe al caso del comando skip, per il quale si è già osservato che non può mai essere interferente. Poiché $v(\mathrm{E},s_1) \neq v(\mathrm{E},s_2)$, si ottiene $\pi_2([\![\mathrm{E}]\!]s) = H$. Per s_2 si ha

  $$[\![\text{while E do } \mathrm{C}_1]\!]_L s_2 = [\![\mathrm{C}_1 \text{ ; while E do } \mathrm{C}_1]\!]_H s_2$$

  Affinché si abbia una situazione di interferenza, in tale comando deve essere eseguito necessariamente almeno un assegnamento di memoria bassa. Il risultato segue dunque dal lemma 5.9.5.

  $n > 0$**:** Il caso in cui $n \neq m$ è banale. In questo caso, difatti, si ottiene esiste un indice i tale che $v(E,s_1^i) \neq v(E,s_2^i)$, e dunque $\forall s.\pi_2(v(E,s)) = H$. Affinché il comando sia interferente, in una delle esecuzioni del ciclo while, partendo dallo stato s_1 oppure dallo stato s_2 si deve avere un assegnamento di memoria bassa, e il risultato segue dal lemma 5.9.4.

  Sia dunque $n = m$. Se all'$n-1$esima iterazione si è ottenuto uno stato di errore, non vi è nulla da dimostrare. Analogamente, se l'esecuzione delle n iterazioni del comando C_1 a partire dalla memoria s_2 restituiscono come valore finale $val_\perp\ \iota_2(*)$, allora ancora una volta non vi è nulla da dimostrare.
  Indicando con s_1^{n-1} ed s_2^{n-1} gli stati ottenuti rispettivamente dopo $n-1$ ed $m-1$ iterazioni del comando C_1 sulle memorie s_1 ed s_2, si deve avere allora

$s_1^{n-1} =_L s_2^{n-1}$. Poiché si è assunto che il comando while E do C_1 è interferente, inoltre, si ha:

$$\mathcal{S}[\![C_1]\!]s_1^{n-1} \not=_L \mathcal{S}[\![C_1]\!]s_2^{n-1}$$

E dunque, per ipotesi induttiva, in almeno uno dei due casi l'interpretazione del comando coincide con lo stato di errore $val_\perp\ \iota_2(*)$.

□

6 Conclusioni e lavoro futuro

In questa tesi si è analizzato uno dei problemi di maggiore rilevanza nell'area di studio della **Language Based Security**.
Tramite un'analisi dettagliata del problema e una formalizzazione rigorosa di un'ampia varietà di strutture matematiche, è stata definita una teoria algebrica in cui le violazioni della politica di sicurezza dell'**information flow** vengono interpretate in degli oggetti matematici, ottenibili quali risultato di una computazione.
Si è però osservato che la teoria sviluppata, sebbene corretta, non è completa. Sebbene i risultati di [DD97] mostrano che non sia possibile ottenere un algoritmo che risolva il problema dell'information flow, non vi è nessun risultato che dimostri che non sia possibile ottenere una teoria algebrica che modelli esattamente la rilevazione dei flussi di informazione: tuttavia, dal risultato sopra citato, se tale teoria esiste, al suo interno saranno contenute delle equazioni la cui soluzione non è calcolabile. Se difatti si assumesse di avere una teoria equazionale per la quale è sempre possibile stabilire tramite un procedimento di calcolo la non interferenza di un comando, tale procedimento potrebbe essere applicato algoritmicamente per decidere il problema dell'information flow. La definizione di una semantica in cui lo stato di errore coincida esattamente con la nozione di interferenza corrisponderebbe a un risultato rilevante, in quanto si avrebbe una formalizzazione completa del problema in termini di una teoria equazionale. Nell'analizzare il problema si è notato che l'interferenza di un comando dipende strettamente dall'insieme degli operatori θ utilizzati nelle espressioni, e dalla teoria definita da tali operatori. Sia dato ad esempio il comando

```
1 l := h − h
```

Sebbene in tale comando venga effettuato un assegnamento esplicito di una espressione in cui compaiono variabili alte in una variabile bassa, è ovvio che tale comando non generi interferenza. Ciò deriva dal fatto che la teoria degli operatori θ stabilisce che il risultato dell'operazione è una costante, indipendentemente dal valore di h. Nel ricercare una semantica che descriva esattamente il problema dell'information flow si deve tenere conto dunque della teoria definita dall'insieme di operatori utilizzati.

Nel caso in cui risulti invece impossibile definire tale semantica, si ritiene comunque che il livello di precisione della teoria equazionale definita in questa tesi possa essere aumentato, in maniera da valutare nello stato di errore una quantità minore di comandi interferenti. Un esempio di come tale obiettivo possa essere ottenuto risiede nel mantenere le informazioni sulle variabili basse che contengono delle informazioni copiate dalla memoria alta: se le informazioni contenute in tali variabili venissero cancellate, difatti, il comando non risulterebbe più interferente. Ad esempio, si consideri il comando

```
1 l := h
```

Utilizzando la semantica definita in questa tesi, indipendentemente dalla configurazione di memoria iniziale si otterrebbe una situazione di errore. Non viene data nessuna informazione su quali variabili contengono dei dati provenienti dalla memoria alta, e pertanto non sarà possibile recuperare da tale errore in comandi più complessi, quali ad esempi

```
1 l := h
2 l := 0
```

Nel lavoro svolto si è considerata la sola tipologia di flusso di informazione corrispondente al trasferimento di informazione parziale da una zona di memoria alta ad una zona di memoria bassa. In letteratura sono stati individuati altri tipi di flussi di informazione, quali i **termination channel**, in cui la nonterminazione di un comando può fornire informazioni sulla zona di memoria alta, o i **timing channel**, in cui tali informazioni sono correlate al tempo necessario per l'esecuzione di un comando.
Poiché nell'affrontare il problema si è scelto di utilizzare le monadi e i costruttori semantici, strutture che permettono un approccio alla semantica denotazionale in maniera modulare, reputiamo che estendere la semantica in modo da tenere conto di nuovi effetti computazionali rappresenti un problema di interesse considerevole. Ovviamente, qualora si voglia estendere l'intepretazione definita ad un linguaggio di programmazione reale, si dovrà tenere conto dei diversi effetti computazionali introdotti dal linguaggio utilizzato; inoltre, si dovrà modificare opportunamente il metalinguaggio, definendo delle primitive computazionali addizionali qualora fosse necessario. Durante la stesura della tesi ci si è interessati soprattutto al caso nel quale più comandi possano essere eseguiti in parallelo: difatti, in [VSI96] vengono esibiti esempi di comandi che risultano essere non interferenti se eseguiti singolarmente, ma provocano dei flussi di informazione quando eseguiti in parallelo con altri processi. Quale esempio di comando che risulti essere non interferente quando eseguito da solo, ma provochi invece una situazione di interferenza se eseguito in parallelo, riportiamo il seguente:

```
1 flag := 0
2 if flag
3     l := h
```

Tale comando è ovviamente non interferente, in quanto il blocco di codice all'interno del while non verrà mai eseguito. Tuttavia, se si considera un secondo comando che viene eseguito concorrentemente al primo, si potrebbe ottenere una situazione di interferenza: sia

```
1  flag := 1
```

Si consideri quindi il seguente scheduling dei comandi:

```
1  flag := 0          //comando 1
2  flag := 1          //comando 2
3      if flag
4          l := h     //comando 1
```

É immediato notare che tale scheduling porta ad una situazione in cui il valore della variabile alta h viene copiato nella variabile bassa l.
Il problema dell'information flow in presenza di esecuzioni concorrenti è stato analizzato durante la stesura della tesi: in [Pap01] viene introdotta la monade delle resumptions, utilizzata per modellare l'**interleaving** delle computazioni; la tripla di Kleisli delle resumptions è costituita da un'associazione di oggetti

$$TA = \mu X.A + X$$

Una computazione sul tipo A può essere espressa come

$$\iota_2(\cdots(\iota_2(\iota_1(a)))\cdots)$$

dove $a : A$. Nell'oggetto $\mu X.T(A + X)$, il morfismo $\iota_1 : A \to \mu X.A + X$ viene inteso alla restituzione di un valore, mentre il morfismo $\iota_2 : A + \mu X.A + X \to \mu X.A + X$ corrisponde alla **esecuzione in un passo atomico**. In [Cen96] viene inoltre mostrato un costruttore semantico per le resumptions, dato da

$$(\mathcal{R}T)A = \mu X.T(A + X)$$

Modellare una computazione in termini di passi atomici non è sufficiente per esprimere la concorrenza quale effetto computazionale: difatti, una computazione parallela potrà restituire diversi risultati in base allo scheduling delle operazioni atomiche. Viene quindi introdotta la monade del nondeterminismo

$$PA = \mathcal{P}A$$

Nella categoria degli insiemi, tale monade associa ad ogni insieme il suo insieme potenza. In [Cen96] viene mostrato che in una categoria cartesiana chiusa in cui sia presente la monade $(RP)A$ è possibile definire un operatore *pand* di composizione parallela, la cui segnatura è data da:

$$pand : ((\mathcal{R}P)A \times (\mathcal{R}P)B) \to ((\mathcal{R}P)(A \times B))$$

Un'ulteriore considerazione sul lavoro effettuato riguarda l'insieme dei contesti di sicurezza utilizzato: per semplicità, si è assunto che tale insieme fosse composto dai soli due elementi L ed H, e che valesse la relazione $L < H$. La semantica definita è facilmente estendibile ad un **lattice** di sicurezza: dato un lattice $(\Sigma, \sqsubseteq)$ di livelli di sicurezza, si dovranno definire le interpretazioni $[\![\cdot]\!]_\sigma$ per ogni $\sigma \in \Sigma$. Nel determinare in quale contesto si dovrà eseguire un eventuale sottocomando, come ad esempio nel caso di un comando del tipo

```
1 while E do C
```

nel quale il contesto σ_1 utilizzato per interpretare il comando e il livello di sicurezza σ_2 dell'espressione determinano in quale contesto verrà eseguito il comando C, si dovrà scegliere il minimo comune maggiorante $\sigma_1 \sqcup \sigma_2$.

Infine, avendo definito la possibilità di operare su delle strutture matematiche tramite l'esecuzione di calcoli algebrici per rilevare i flussi di informazione, reputo personalmente utile implementare la teoria ottenuta all'interno di un **proof assistant**: tale passo, oltre a fornire uno strumento per la rilevazione di flussi di informazione tramite l'interazione con un elaboratore, permetterebbe inoltre di identificare diverse tattiche di dimostrazione che possono contribuire allo sviluppo della teoria.
In conclusione, durante la stesura del presente lavoro sono stati individuati diverse strategie per lo sviluppo futuro del lavoro effettuato:

- Definizione di una teoria algebrica che modelli in maniera **Corretta** e **Completa** il problema dell'information flow. Inoltre, sarebbe di rilevante interesse riuscire a modellare il concetto di **termination channel** in tale teoria.

- Scalabilità a linguaggi di programmazione con maggiori effetti computazionali: in particolare, si è interessati ad estendere la teoria sviluppata al caso in cui sia possibile eseguire più comandi in parallelo. In [Pap01] viene analizzato l'effetto computazionale delle resumptions, tramite il quale si modellano le computazioni parallele in termini di **interleaved executions**. Il costruttore semantico $(\mathcal{R}T)A = \mu X.T(A + X))$ può essere utilizzato per estendere la teoria sviluppata.

- Implementazione della teoria all'interno di un **proof assistant**, analisi dei procedimenti di dimostrazione e definizione di tattiche di dimostrazione personalizzate che siano di aiuto all'utente nell'analisi della sicurezza dei programmi.

So long and thanks for all the fish.

Douglas Adams,
Hitchickers' Guide to Galaxy

Bibliografia

[AL91] Andrea Asperti and Giuseppe Longo. *Categories, Types and Structure.* M.I.T Press, 1991.

[Bar92] Henk P. Barendregt. *Lambda Calculi with Types.* Handbook of Logic in Computer Science, Volume II, 1992.

[Bis03] Tom Bishop. *Computer Security: Art and Science.* Addison-Wesley, 2003.

[CC08] S. Cavadini and D. Cheda. *Run-time Information Flow Monitoring based on Dynamic Dependence Graphs.* Availability, Reliability and Security, 2008. ARES 08. Third International Conference on Volume, 2008.

[Cen96] Pietro Cenciarelli. *Computational Applications of Calculi based on Monads.* University of Edinburgh, Department of Computer Science, 1996.

[Cen99] Pietro Cenciarelli. *Towards a modular denotational semantics for Java.* Formal Techniques for Java Programs, Proceedings, Lisbon, Portugal, 1999.

[CHM05] David Clark, Sebastian Hunt, and Pasquale Malacaria. *Quantitative information flow, relations and polymorphic types.* Journal of Logic and Computation, Oxford Journals, 2005.

[DD97] Dorothy E. Denning and Peter J. Denning. *Certification of Programs for Secure Information Flow.* 1997.

[GS08] Daniele Gorla and Ivano Salvo. *Dispense del corso di Tecniche di Sicurezza Basate sui Linguaggi.* 2008.

[HH05] William L. Harrison and James Hook. *Achieving Information Flow Security Through Precise Control of Effects.* 2005.

[HMS96] Kevin W. Hamlen, Greg Morrisett, and Fred B. Schneider. *Computability Classes for Enforcement Mechanisms.* 1996. http://www.cs.cornell.edu/fbs/publications/EnfClassesTR2003-1908.pdf.

[Lan98] Saunders Mac Lane. *Categories for the Working Mathematician.* Springer, 1998.

[LHJ95] Sheng Liang, Paul Hudak, and Mark Jones. *Monad transformers and modular*

interpreters. Annual Symposium on Principles of Programming Languages, 1995.

[Mog89a] Eugenio Moggi. *An Abstract View of Programming Languages*. 1989.

[Mog89b] Eugenio Moggi. *Computational lambda calculus and monads*. 1989.

[Mog91] Eugenio Moggi. *A Modular Approach to Denotational Semantics*. Lecture Notes in Computer Science, Springer Berlin/Heidelberg, 1991.

[MSZ04] A. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification, 2004.

[Pap98] Nikolaos S. Papaspyrou. *A formal Semantics for the C Programming Language*. National Technical University of Athens, Department of Electrical and Computer Engineering, 1998.

[Pap01] Nikolaos S. Papaspyrou. *A Resumption Monad Transformer and its Applications in the Semantics of Concurrency*. 2001.

[Pie91] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists*. M.I.T. Press, 1991.

[Sch03] Fred B. Schneider. *Enforceable Security Policies*. 2003. http://www.cs.cornell.edu/fbs/publications/EnfSecPols.pdf.

[Sip97] Michael Sipser. *Introduction to the theory of Computation*. PWS Publishing Company, 1997.

[SM03] Andrei Sabelfeld and Andrew C. Myers. *Language based Information-Flow Security*. 2003.

[SV98] Geoffrey Smith and Dennis Volpano. *Secure Information-flow in a Multi-threaded Imperative Language*. Annual Symposium on Principles of Programming Languages, 1998.

[TN02] Allen Tucker and Robert Noonan. *Programming Languages, Principles and Paradigms*. McGraw Hill Higher Education, 2002.

[VBC+04] N. Vachharajani, M.J. Bridges, J. Chang, R. Rangan, G. Ottoni, J.A. Blome, G.A. Reis, M. Vachharajani, and D.I. August. *RIFLE: An Architectural Framework for User-Centric Information-Flow Security*. 2004.

[VSI96] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. *A sound type system for secure flow analysis*. 1996.

[Wad90] Philip Wadler. *Comprehending Monads*. 1990.