Alma Mater Studiorum - Università di Bologna

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
Corso di Laurea Triennale in Informatica

SEMPLIFICARE L'INTERAZIONE CON L'UTENTE NELL'INTERPRETE INTML

Tesi di Laurea in Sicurezza e Crittografia

Relatore:
Ill.mo Dott. Ugo Dal Lago

Presentata da:
Federica Foschi

I Sessione
Anno Accademico 2012/2013

Indice

1 Introduzione **5**

2 Computazione Limitata in Spazio **9**

2.1 Macchine di Turing 11

2.2 Macchina di Turing Offline 12

2.3 Classi di Complessità 14

2.4 Programmazione Bidirezionale 15

3 Il Paradigma Funzionale **17**

3.1 Linguaggi Funzionali e Imperativi a Confronto 17

3.2 IntML 20

3.2.1 Working Class 21

3.2.2 Upper Class 24

3.2.3 Compilazione da UC a WC 27

3.3 OCaml 29

4 L'interprete per IntML **33**

4.1 Interpreti e Compilatori 33

4.2 L'interprete per IntML 35

4.2.1 Regole per la valutazione dei termini 35

5 Esplorazione di Dati Strutturati **37**

5.1 Migliorare l'Interazione con l'Interprete 38

5.2 Stringhe 39

5.2.1 Esempi di Stringhe 40
5.2.2 Stringhe della WC e UC a Confronto 43
5.2.3 #explore *string* 46
5.2.4 Test 53
5.3 Alberi Binari 55
5.3.1 Esempi di Alberi 58
5.3.2 Alberi della UC e WC a confronto 60
5.3.3 Alberi e Grafi: Topsort 61
5.3.4 #explore *tree* 64
5.3.5 Test 72
5.4 Codice Completo di #explore 74
5.5 Note sulla Realizzazione del Codice di #explore 88

Capitolo 1

Introduzione

Lo sviluppo tecnologico ha portato negli ultimi decenni a una continua evoluzione dei sistemi di calcolo, a livello sia di potenza dei processori che di capacità e velocità dei dispositivi di memorizzazione. L'esistenza di hardware sempre più efficiente si accompagna a tecniche sempre più sofisticate che permettono l'ottimizzazione del tempo di calcolo del processore (sistemi multiprocessore, scheduling della CPU) e delle interazioni con la memoria principale e secondaria (scheduling dei dischi, cache e memoria virtuale, TLB ecc.). La disponibilità di processori sempre più potenti e di dispositivi di memorizzazione più ampi ed efficienti conduce in modo naturale a progettare applicazioni via via più complesse, che tendono a sfruttare in maniera pervasiva la tecnologia disponibile, mentre l'utilizzo di tecniche come la memoria virtuale solleva il programmatore dal problema dei limiti fisici della memoria, dandogli l'illusione di disporre di uno spazio di memorizzazione molto più ampio di quello effettivamente disponibile sul sistema di calcolo.

L'approccio più immediato nell'elaborazione di un algoritmo o di un programma è in genere quello che privilegia l'aspetto del tempo impiegato dalla computazione, piuttosto che dello spazio. Il programmatore che ricerca l'ottimizzazione del tempo di calcolo sa che il suo algoritmo, dato un certo input, dovrà produrre un determinato output dopo un tempo t calcolabile come funzione della dimensione dell'input. Sappiamo dalla teoria della calcolabilità

che alcuni problemi sono *intrinsecamente difficili*, il che significa che possono essere risolti soltanto in un tempo esponenziale rispetto alla dimensione dell'input, mentre per altre tipologie di problemi esistono molteplici algoritmi risolutivi, alcuni dei quali lavorano in tempo lineare o pseudolineare. Cosa succede, tuttavia, quando la dimensione stessa dell'input ha dimensioni enormi, al punto di non poter nemmeno essere contenuta nella memoria di un singolo calcolatore? Lo spazio richiesto dalla computazione svolge un ruolo cruciale in molte applicazioni sviluppatesi negli ultimi decenni: da un lato web searching, data mining e tutte le tipologie di applicazioni che devono accedere a dati di dimensione potenzialmente infinita, dall'altro l'impiego sempre più pervasivo dei mircocontrollori, che pur avendo processori e memorie dalle dimensioni ridotte devono garantire una serie sempre crescente di funzionalità. E' dunque naturale che parte della ricerca sia rivolta a mitigare il problema dello spazio occupato dalla computazione, tramite l'utilizzo di algoritmi appositi o di linguaggi di programmazione progettati *ad hoc*.

La difficoltà principale incontrata dal programmatore abituato a lavorare con linguaggi tradizionali imperativi e a ragionare in termini di tempo della computazione nasce dal fatto che, per pensare in termini di spazio, occorre utilizzare un diverso paradigma - la programmazione bidirezionale, come chiariremo in seguito – e linguaggi funzionali che forniscono un approccio più matematico, ma meno inutitivo, per la creazione di algoritmi efficienti. Per ovviare a questo problema è opportuno fornire al programmatore degli strumenti per lavorare nel modo più naturale possibile, ovvero delle tecniche che mascherino la complessità del sistema sosttostante. Un linguaggio di recente introduzione, IntML, si propone di superare questa difficoltà fornendo al programmatore delle primitive che consentono di elaborare algoritmi in modo naturale, poichè è il linguaggio stesso a occuparsi della traduzione da approccio unidirezionale a bidirezionale.

IntML è descritto a livello teorico in [DLS10a] ma possiede anche una implementazione concreta, grazie all'esistenza di un interprete – ancora in fase prototipale – che permette di realizzare programmi che lavorano in spazio

logaritmico rispetto alla dimensione dell'input. In questo lavoro apportiamo delle migliorie nella interazione tra interprete e programmatore: in particolare, elaboreremo un algoritmo che consente all'interprete IntML di valutare in automatico e stampare a schermo il valore di funzioni che rappresentano stringhe e alberi binari, operazione che in precedenza doveva essere eseguita manualmente dall'utente bit a bit.

Dopo una breve introduzione sulle classi di complessità legate a tempo e spazio, indispensabile punto di partenza teorico per la comprensione di IntML e dei principi della programmazione bidirezionale (cap. 2), ci occuperemo del paradigma funzionale e in particolare di IntML e di OCaml, il linguaggio in cui è scritto l'interprete (cap. 3). Descriveremo in seguito il funzionamento dell'interprete allo stato attuale della ricerca (cap. 4) e presenteremo infine i miglioramenti apportati (cap. 5).

Capitolo 2

Computazione Limitata in Spazio

La teoria della complessità è un ramo dell'Informatica Teorica che studia la classificazione dei problemi computazionali in base alla loro inerente difficoltà, ovvero alle risorse necessarie per la loro soluzione. Tali risorse sono principalmente il tempo e lo spazio necessari alla computazione, sebbene possano essere definite ulteriori metriche dipendenti dal particolare problema preso in considerazione. L'efficienza di un algoritmo deve essere misurata tenendo presente tutti i possibili input ammissibli, ovvero è data tenendo conto del caso pessimo (quello in cui la particolare strutturazione dei dati in input richiede all'algoritmo il maggior numero di passi di computazione) e deve prescindere dal particolare elaboratore impiegato. Per questo motivo è necessario definire un modello di calcolo indipendente dalle specifiche tecnologie utilizzate: adotteremo in questo contesto il modello delle Macchine di Turing (d'ora in poi, MdT), sebbene sia possibile utilizzare modelli alternativi quali ad esempio il λ-calcolo. E' possibile dimostrare che la MdT, pur essendo concettualmente semplice, è in grado di risolvere qualsiasi problema che sia algoritmicamente calcolabile; tutti i moderni linguaggi di programmazione (Java, C, ML...) hanno lo stesso potere espressivo della MdT e sono

perciò detti Turing-completi o Turing-equivalenti[1]
Fra i problemi a cui è possibile dare una soluzione algoritmica alcuni sono facili, altri sono inerentemente difficili, a seconda del tempo o dello spazio necessario alla computazione: tempo e spazio sono calcolabili in maniera precisa (a meno di costanti) come funzione della dimensione dei dati in input. Possiamo così dire, in via generale, che dato un input di dimensione n un problema facile ammette una soluzione che preveda un numero di passi di computazione al più polinomiale $\mathcal{O}(n^c)$ per un qualche c costante (anche se, intuitivamente, per c grandi la computazione diviene di fatto non realizzabile), mentre un problema difficile prevede una soluzione esponenziale $\mathcal{O}(c^n)$, $c > 1$. Occorre inoltre fare una netta distinzione tra il *problema* e l'*algoritmo* impiegato per risolverlo: per fare un esempio, il problema dell'ordinamento di un array è univoco e ben definito, ma per risolverlo sono stati progettati numerosi algoritmi, alcuni che lavorano in tempo $\mathcal{O}(n^2)$ – bubble sort, insertion sort – e altri in tempo $\mathcal{O}(nlogn)$ – mergesort, quicksort nel caso medio. Di conseguenza, definire un problema come inerentemente difficile significa affermare che non è possibile fornire un algoritmo al più polinomale che lo risolva. Questo è dovuto da un lato alla difficoltà caratteristica di certi tipi di problemi, dall'altro alla natura finitistica dei mezzi a supporto della computazione.
Fatta questa premessa, vogliamo brevemente introdurre alcune fra le principali classi di complessità, focalizzando l'attenzione su quelle che maggiormente riguardano il nostro lavoro e le performance garantite da IntML. Partiamo dunque la definizione formale delle MdT, sia nella versione base sia nel cosidetto modello delle MdT Offline (d'ora in poi MTO), che come vedremo sono il necessario supporto per lo studio della complessità in termini di spazio; presenteremo poi alcune fondamentali classi di complessità, utili per il nostro

[1] Scrivere un algoritmo nel linguaggio della MdT risulterebbe alquanto tedioso, ma è facile convincersi della riducibiltà di un linguaggio ad alto livello a quello della MdT: basta pensare al fatto che qualsiasi programma una volta compilato è tradotto in una serie di istruzioni assembly che si riducono, in ultima istanza, al caricamento e salvataggio di dati nelle celle di memoria, esattamente come fa la MdT.

lavoro, e infine introdurremo il modello della computazione bidirezionale.

2.1 Macchine di Turing

La MdT fu introdotta da Alan Turing a metà degli anni '30: si tratta di un modello di calcolo di fondamentale importanza sia per la teoria della calcolabilità sia per il mondo dell'informatica in generale, in quanto l'architettura di von Neumann, progettata negli anni '40, nacque propro dall'intuizione che il modello astratto della MdT poteva essere realizzato concretamente.
Nella sua versione base, la MdT è costituita da $k \geq 1$ nastri di memoria illimitati, divisi in celle di dimensione fissata. Le celle possono contenere un carattere qualsiasi di un alfabeto finito prefissato oppure uno speciale carattere *blank* che indica la non inizializzazione. Ad ogni nastro corrisponde una testina, che in un passo di computazione può spostarsi sulla cella a sinistra o a destra rispetto alla posizione attuale. Un automa a stati finiti (il programma della macchina) decide, a seconda della posizione e del carattere letto, la successiva azione da eseguire. Le azioni compiute dalla macchina sono elementari: ad ogni passo di computazione la testina legge un carattere, l'automa decide l'azione successiva, la testina si sposta e scrive un nuovo carattere. Nel caso di macchine a più nastri ($k \geq 2$) il primo nastro è riservato all'input e l'ultimo all'output; gli eventuali nastri intermedi sono considerati nastri di lavoro. All'inizio della computazione il nastro di input contiene n caratteri (la dimensione dell'input) e la testina k_1 si trova sulla prima posizione. Formalmente, una MdT multi-nastro deterministica è una tupla $(Q, \Gamma, b, \Sigma, k, \delta, q_0, F)$ dove:

- Q è un insieme finito di stati;
- Γ è l'alfabeto finito del nastro;
- b è il carattere *blank*;
- $\Sigma \subseteq \Gamma \setminus \{ b \}$ è l'insieme dei caratteri di input/output;
- $k \geq 1$ è il numero dei nastri;
- $q_0 \subseteq$ Q è lo stato iniziale della macchina;

- $F \subseteq Q$ è l'insieme degli stati finali;
- δ = Q \F x $\Gamma_k \rightarrow$ Q x Γ_k x (L,R)$_k$ è la funzione di transizione (L = left, R = right)

Nel modello deterministico δ è definita come funzione, ossia ad una particolare configurazione istantanea della macchina (data dallo stato Q_i, la posizione delle k testine e il contenuto del nastro) corrisponde una ed una sola azione per ogni testina. Nella MdT non deterministica (MTN), invece, δ è una relazione, e dunque ogni configurazione ammette più continuazioni possibili (eventualmente zero), dando origine ad una computazione non più lineare ma strutturata ad albero.
La MdT termina la computazione se raggiunge una configurazione di accettazione o di rigetto (di conseguenza, accettando o rifiutando la stringa data in input), entra in loop in caso contrario; con la MTN, invece, la stringa in input è accettata se almeno un ramo di computazione fra tutti quelli possibili conduce ad una configurazione di accettazione, è rifiutata se tutti conducono ad una configurazione di rigetto.
Considerando che la MdT accetta in input una stringa e ne produce un'altra in output, possiamo vederla come una funzione f: $\Sigma^* \rightarrow \Sigma^*$[2]. Estendiamo questa definizione dicendo che l'insieme delle stringhe accettate da una MdT (ossia, che fanno terminare la computazione in uno stato di accettazione) è il linguaggio riconosciuto da quella determinata macchina.

2.2 Macchina di Turing Offline

La MTO è un modello alternativo di MdT impiegato per definire formalmente gli algoritmi con complessità sublineare in spazio. La MTO è una MdT multinastro con $k > 2$ in cui il primo nastro è di solo input e read-only, l'ultimo è di solo output e write-only, e i nastri intermedi sono nastri di lavoro. Nei nastri 1 e k la testina può soltanto avanzare verso destra. La classe di complessità $\mathbb{SPACE}$ a cui appartiene la macchina si calcola considerando

[2]La stella di Kleene indica tutte le successioni di lunghezza finita di elementi di Σ: se Σ è un alfabeto, rappresenta tutte le infinite stringhe realizzabili su di esso.

la quantità di celle utilizzate durante la computazione ovvero, trattandosi di ordini di grandezza, valutando il nastro col maggior numero di celle inizializzate. La particolare struttura della MTO - che non ha bisogno di memorizzare nè il proprio input (è read-only e viene letto da un dispositivo esterno) nè l'output (è prodotto come stream di caratteri) – permette di considerare in questo calcolo soltanto i nastri di lavoro. E' facile convincersi infatti che, se dovessimo tener conto anche delle n celle inizializzate nel nastro che contiene l'input, non potremmo mai sperare di raggiungere una complessità sublineare in spazio. La macchina avrà accesso all'input non nella sua totalità ma una cella alla volta e *su richiesta*, ossia tramite interazione con l'ambiente. Possiamo dare una descrizione intuitiva del funzionamento della MTO:

- se la macchina riceve dall'ambiente un input n, significa che deve calcolare l'n-simo carattere del suo output;

- se la macchina necessita di leggere l'm-simo carattere dell'input, produce in output una coppia (s, m) dove s è la configurazione istantanea della macchina, necessaria per riprendere la computazione da dove era stata interrotta, e m è l'indice del carattere richiesto; l'ambiente risponderà producendo a sua volta una coppia (s, i_m) dove s è lo stato salvato della macchina e i_m è il carattere richiesto.

Possiamo interpretare la MTO come una funzione $(State \text{ x } \Sigma) + \mathbb{N} \rightarrow (State \text{ x } \mathbb{N}) + \Sigma$ dove il simbolo + rappresenta l'unione disgiunta dei due insiemi. La definizione delle MTO, oltre a dare supporto teortico alla definizione di algoritmi con complessità sublineare in spazio, costituisce il punto di partenza del passaggio dalla comunicazione unidirezionale a quella bidirezionale o su richiesta. Pensiamo ad esempio di voler realizzare la composizione di due Macchine di Turing, utilizzando rispettivamente MdT e MTO. Possiamo fare le seguenti osservazioni:

- **MdT**: per ottenere il risultato finale le due macchine devono essere lanciate sequenzalmente, poichè la seconda deve attendere l'output ge-

nerato dalla prima, realizzando così una comunicazione unidirezionale. Le dimensioni del risultato intermedio non sono note a priori;

- **MTO**: per ottenere ogni carattere di output della seconda macchina, questa richiederà alla prima l'input necessario per il calcolo, avviando quindi la computazione sulla prima macchina e realizzando un flusso di dati bidrezionale. Il risultato intermedio non viene generato nella sua totalità e le sue dimensioni sono limitate da un bound noto a priori.

2.3 Classi di Complessità

Vogliamo ora introdurre alcune fra le principali classi di complessità. Data una MdT M definiamo:

- $time_M(x)$: il tempo richiesto da M per la computazione sull'input x (ossia il numero di passi di computazione necessari)
- $t_M(x) = \max \{ \ time_M(x) : | \ x \ | = \text{n} \ \}$
- $space_M(x)$: il numero massimo di celle visitate da una qualche testina di M (considerando soltanto i nastri di lavoro)
- $s_M(x) = \max \{space_M(x) : | \ x \ | = \text{n} \ \}$

Data una funzione f: $\mathbb{N} \rightarrow \mathbb{N}$ possiamo quindi definire le due classi di complessità deterministica:

- $\mathbb{DTIME}(\text{f}) = \{ \ \text{L} \subseteq \Sigma^* \ / \ \exists M : L = L_M, t_M \subseteq O(f) \ \}$
- $\mathbb{DSPACE}(\text{f}) = \{ \ \text{L} \subseteq \Sigma^* \ / \ \exists M : L = L_M, s_M \subseteq O(f) \ \}$

Esse rappresentano l'insieme dei linguaggi per cui esiste una MdT che li riconosce, e che termina la sua computazione in tempo/spazio $\mathcal{O}(f)$. In particolare:

- $\mathbb{LOGSPACE} = \mathbb{DSPACE}(log)$
- $\mathbb{PSPACE} = \bigcup_{c \subseteq \mathbb{N}} \mathbb{DSPACE}(n^c)$
- $\mathbb{EXPSPACE} = \bigcup_{c \subseteq \mathbb{N}} \mathbb{DSPACE}(2c^n)$

Per le MTN possiamo dare definizioni analoghe, tenendo presente che per valutare $time_M(x)$ e $space_M(x)$ considereremo un ramo di computazione generico fra tutti quelli possibili. Le corrispondenti classi di complessità sono $\mathbb{NTIME}(f)$ e $\mathbb{NSPACE}(f)$.

2.4 Complessità Sublineare e Programmazione Bidirezionale

Fra tutte le classi di complessità considerate, quelle che assumono maggiore rilievo nell'ambito del nostro lavoro sono le classi con complessità sublineare: appartengono a questa categoria gli algoritmi che impiegano per la computazione uno spazio di memoria inferiore a quello necessario per memorizzare il loro input, e che si mostrano particolarmente utili qualora sia necessario manipolare input così grandi da non poter essere contenuti nella memoria dell'elaboratore (pensiamo ad esempio a ricerche su un grande database). Scrivere algoritmi che rispettino questi bound spaziali non è semplice, e richiede l'utilizzo di particolari tecniche che, se non gestite correttamente, possono facilmente condurre ad errori di programmazione, e che rendono complicata la progrettazione di procedure altrimenti semplici. Torniamo all'esempio della composizione due funzioni (2.2): se vogliamo che la complessità della composizione resti sublineare non possiamo computare le due funzioni sequenzialmente poichè, in via generale, il risultato intermedio potrebbe avere una dimensione che eccede i limiti di memoria a disposizione. Una soluzione al problema può essere fornita dalla tecnica della programmazione bidirezionale, o programmazione su richiesta, che prevede che il risultato intermedio non sia calcolato nel suo complesso, ma piuttosto che vengano memorizzate di volta in volta solo piccole parti di esso, e che le stesse vengano eventualmente ricalcolate in caso di necessità. Detto il altre parole, lo stile bidirezionale prevede che il flusso delle informazioni segua due direzioni: da un lato, l'ambiente inoltra al programma richieste di (parti di) output, dall'altro il programma inoltra all'ambiente richieste di (parti di) in-

put. La computazione procede tramite un continuo scambio di messaggi, e quindi tramite la costruzione di una rete di message-passing, con cui ambiente e programma comunicano scambiandosi un bit di informazione alla volta. Vale la pena mettere in evidenza che in questo modello di computazione una macchina deve essere pronta a ricevere più volte la stessa richiesta e calcolare *ex novo* il valore del bit domandato: tenere traccia dei bit già calcolati, infatti, farebbe velocemente superare il limite di occupazione sublineare della memoria che ci siamo imposti. Questo significa che i nodi che implementano la rete di message-passing realizzano una computazione *stateless*, ossia non possono avere memoria della computazione già effettuata[3].

Dal punto di vista del programmatore, implementare lo scambio di messaggi per il calcolo dei risultati intermedi potrebbe rivelarsi un compito gravoso: per questo motivo è auspicabile avere a disposizione un meccanismo che nasconda la complessità del modello sottostante, permettendo di progettare gli algoritmi nel modo usuale e incaricandosi in automatico del passaggio da stile unidirezionale a stile bidirezionale.

Si può quindi ragionevolmente pensare di progettare un linguaggio di programmazione arricchito con speciali *features* che permettano questa traduzione da uno stile all'altro, seguendo due approcci fondamentali: nel primo, il linguaggio maschera completamente il modello sottostante, incaricandosi della traduzione - ad esempio in fase di compilazione - mentre nel secondo il linguaggio fornisce primitive per lavorare con la computazione su richiesta, le quali possono essere usate liberamente dal programmatore solo in caso di bisogno. Il secondo approccio si mostra ragionevole pensando al carico computazionale dovuto all'implementazione degli algoritmi con complessità $\mathbb{LOGSPACE}$, che in generale non costituiscono l'intero programma, e dunque non devono ostacolare l'esecuzione delle parti del codice che non necessitano di queste funzionalità aggiuntive.

[3]Parlando delle MTO avevamo visto che la macchina non ha modo di ricordare il suo stato, ma lo produce in ouput quando richiede all'ambiente una determinata parte di input, e lo riceve poi in input – insieme al valore richiesto – in modo da sapere da dove ricominciare la computazione.

Capitolo 3

Il Paradigma Funzionale

In questo capitolo illustreremo le principali caratteristiche del paradigma funzionale, a cui appartengono i due linguaggi di programmazione oggetto di questo lavoro, IntML e OCaml, e descriveremo in modo dettagliato i costrutti di IntML.

3.1 Linguaggi Funzionali e Imperativi a Confronto

Il paradigma funzionale trae fondamento teorico dal λ-calcolo, sviluppatosi negli anni '30 a fianco del modello delle macchine di Turing, da cui invece trae origine il paradigma imperativo; al pari di quest'ultimo si tratta di una famiglia di linguaggi Turing-completi, ossia in grado di esprimere tutte le funzioni algoritmicamente calcolabili.
La caratteristica fondamentale dei linguaggi funzionali è, almeno nella loro versione pura, quello di non possedere il concetto di memoria: la computazione procede tramite riscrittura di valori, con modifiche che hanno luogo solo nell'ambiente. Perde dunque senso il concetto di variabile modificabile e, di conseguenza, il costrutto di assegnamento su cui è basato il paradigma imperativo. Il paradigma funzionale si accompagna in modo naturale al concetto di ricorsione, suo fondamentale costrutto per il controllo di sequenza.

Un programma funzionale computa espressioni, producendo valori. L'ordine con cui vengono eseguite le operazioni non è importante (non ci sono effetti collaterali), e il programmatore non deve curarsi della rappresentazione fisica dei dati in memoria, visto che non può accedere ad essa in modo diretto, nemmeno per deallocare i dati stessi (operazione demandata in automatico a un garbage collector). Un programma imperativo, invece, procede eseguendo sequenzialmente operazioni che modificano lo stato della memoria, e il programmatore si trova spesso a manipolare esplicitamente i puntatori ai dati. I linguaggi imperativi sono dunque concettualmente più vicini all'hardware della macchina, ma producono programmi potenzialmente meno sicuri in fase di esecuzione; i linguaggi funzionali, fornendo un più alto livello di astrazione e impiegando un sistema di tipizzazione solitamente più severo, eliminano la possibilità di accedere a valori incoerenti in fase di esecuzione, e di fatto rendono i programmi pià sicuri. Tale modello presenta però alcuni svantaggi: da un lato, il programmatore tradizionale può trovare difficoltà ad esprimersi in questo paradigma, più matematico e astratto; dall'altro, alcune tipologie di algoritmi sono per loro stessa natura più facilmente esprimibili con construtti imperativi. Per questo motivo la maggior parte dei linguaggi funzionali prevede la possibilità di utilizzare anche costrutti di tipo imperativo e di dichiarare e modificare variabili.

Da un punto di vista sintattico, un linguaggio funzionale puro non ha comandi (dato che non è possibile modificare lo stato per effetto collaterale) ma solo espressioni. Partendo dal set convenzionale dei dati primitivi (interi, caratteri, stringhe, booleani...) e delle operazioni per manipolarli da un lato, e dall'espressione condizionale dall'altro, è possibile creare nuove espressioni tramite due costrutti principali, astrazione e applicazione:

Astrazione: function p → exp Un'espressione funzionale è costituita da un parametro (il nome di una variabile) e da un corpo (l'espressione che segue la →). Il parametro formale è detto astratto, motivo per cui l'espressione

funzionale può essere chiamata astrazione.

Applicazione: (function p $\to$ exp) par L'applicazione della funzione all'argomento è ottenuta scrivendo la funzione seguita dal suo argomento. La valutazione dell'applicazione viene eseguita valutando il corpo della funzione dopo aver sostituito il parametro formale con quello attuale.

La possibilità di impiegare funzioni higher-order senza vincoli consente una perfetta omogeneità tra programmi e dati: e funzioni sono valori esprimibili, ossia possono essere il risultato della valutazione di una espressione complessa.
Il corpo di un'espressione funzionale può contenere un'espressione qualsiasi, e quindi anche una o più funzioni come nel seguente esempio (le parentesi non sono necessarie)[1]:

```
# function x -> (function y -> 3*x + y);;
: int -> int -> int = <fun>
```

Possiamo interpretare il tipo di questa espressione in maniera tradizionale come quella di una funzione che aspetta due interi e restituisce un intero, ma nell'ambito di un linguaggio funzionale come OCaml è più corretto (sebbene equivalente) considerarla come una funzione che aspetta un intero e restituisce un valore funzionale di tipo $int \to int$.
Un valore funzionale, ossia una chiusura (tripla costituita da nome del parametro funzionale, corpo della funzione e ambiente di valutazione), può essere restituito come risultato o passato come argomento di un'altra funzione; la funzione h qui di seguito ha come parametro una funzione da intero a intero e restituisce un valore funzionale di tipo int $\to$ int:

```
# let h = function f -> function y -> (f y) + y;;
: val h : (int -> int) -> int -> int = <fun>
```

[1]In questo e nei seguenti esempi impieghiamo la sintassi di OCaml: la prima riga è il prompt dei toplevel interattivo del linguaggio, la seconda è il risultato della valutazione dell'espressione da parte dell'interprete.

Nei linguaggi funzionali un nome di variabile legato ad un valore tramite una dichiarazione mantiene tale valore fino alla fine della sua esistenza; si può spezzare tale legame tramite una ridefinizione, che in realtà non modifica la variabile esistente ma ne crea un'altra, con lo stesso nome, in un'altra locazione di memoria non nota al programmatore. La nuova variabile maschera la precedente, che diventa non più accessibile, ma il cui valore è immutato; l'area di memoria non più utilizzata sarà recuperata dal garbage collector.

3.2 IntML

IntML è un linguaggio funzionale basato sulla costruzione Int di Joyal, Street e Verity. Come si dimostra in [DLS10a], l'applicazione di questo modello permette di creare le reti di message passing necessarie per realizzare la computazione bidirezionale limitata in spazio. Un programma scritto in IntML è composto da funzioni che trasmettono dati tramite canali di comunicazone bidirezionali. Ogni funzione è un nodo della rete, connesso da uno o più cavi con altri nodi o con l'ambiente. Ogni cavo X entrante o uscente da un nodo rappresenta un flusso di dati che può circolare in due direzioni, una che ha lo stesso orientamento del cavo (e che etichettermo con un tipo X^+, che rappresenta gli input per la funzione) e una che ha quella opposto (tipo X^-, che rappresenta i possibili output).
IntML è dotato di due classi di termini e tipi. Per la sua definizione si parte da un normale linguaggio funzionale, che ne costituisce la Working Class (WC): per essa, a motivo di semplicità, si è scelto un linguaggio del prim'ordine con tipi finiti. L'utilizzo dei termini e tipi della WC rende già possibile l'implementazione della comunicazione bidirezionale, ma con le ovvie difficoltà e possibili errori legati a questo stile di programmazione. E' stata quindi introdotta la Upper Class (UC), coi relativi termini e tipi, che fornisce al programmatore delle primitive per progettare le reti di message passing in modo più naturale. Queste primitive sono estensioni conservative della WC, ossia permettono di scrivere gli stessi circuti che si sarebbero potuti pro-

gettare con la WC, ma in modo molto più semplice; ogni termine della UC prima di essere valutato dall'interprete deve essere compilato in termini della WC. Esiste una precisa corrispondenza tra le funzioni calcolabili da IntML e la classe $\mathbb{LOGSPACE}$: per le dimostrazioni formali rimandiamo al cap. 5 di [DLS10a].
IntML possiede una implementazione concreta, grazie all'esistenza di un interprete sul cui funzionamento torneremo con maggior dettaglio nel cap. 4. L'interprete richiede che le espressioni da valutare siano contenute in un file con estensione *.iml* e che rispettino le seguenti regole:

- i termini della WC devono essere introdotti dall'operatore `=W`
- i termini della UC devono essere introdotti dall'operatore `=U`
- le espressioni devono terminare con `;`
- i commenti vanno racchiusi tra `(* ... *)`
- la valutazione di un termine della WC avviene digitando il suo nome dopo il prompt
- la valutazione di un termine della UC avviene digitando il suo nome dopo il prompt seguito da `<>`

3.2.1 Working Class

I tipi della WC sono del prim'ordine con variabili di tipo. La cardinalità di tutti i tipi (intesi come insieme di valori possibili) è finita:

A, B ::= 'α | 1 | A x B | A + B

- **'**α è una variabile di tipo, ossia un tipo generico non ancora istanziato
- **1** è il singoletto, ossia un qualsiasi tipo che contiene un solo elemento
- **A x B** rappresenta il prodotto cartesiano dei tipi A, B; la sua cardinalità è $\mid A \mid * \mid B \mid$

- **A + B** rappresenta l'unione disgiunta dei tipi A, B; la sua cardinalità è $\mid A \mid + \mid B \mid$

I termini della WC sono definiti dalla seguente grammatica (dove le lettere c, d indicano variabili, e f, g, h termini della WC):

f, g, h ::= c | * | min_A | succ_A(f) | eq_A(f,g) | inl(f) | inr(f) | case f of inl (c) → g | inr(d) → h | $< f, g >$ | fst(f) | snd(f) | loop (c.f)(g) | unbox

- **Singleton** Il singoletto `*` si indica con `<>` ed è l'unico valore presente nel tipo 1

- **Operatori costanti** Le costanti `min succ eq`, di ovvia semantica, definiscono per ogni tipo una relazione di ordine totale. Nell'esempio che segue creiamo due termini appartenenti a un tipo con cardinalità 2, che identificheremo con i booleani *true* e *false* (la scelta più naturale è quella di un tipo 1 + 1). Seguendo l'impostazione dell'interprete, assegnamo a *true* il valore 0 e a *false* 1, e quindi istanziamo *true* come l'elemento minimo del tipo 1 + 1, e *false* come il suo successore. All'atto di invocare `min` applichiamo una coercizione (indicata da :) per forzare l'algoritmo di type inference a utilizzare il tipo specifico da noi indicato (le coercizioni possono essere applicate alla costante *min* della WC e a qualsiasi tipo della UC):

  ```
  true =W min: 1+1;     false =W succ true;
  ```

  Valutando i termini con l'interprete otterremo il seguente risultato:

  ```
  # true                # false               # eq true false
  : 1 + 1 = inl(<>)     :1 + 1 = inr(<>)      : 1 + 1 = inr(<>)
  ```

- **Iniezione destra e sinistra** I termini `inl(t)` e `inr(t)` permettono di denotare un elemento di tipo A + B mantenendo l'informazione sull'insieme originario

- **Coppie** Per creare una coppia si utilizza il costruttore `< f, g >`. Gli operatori `fst(f) snd(f)` non sono predefiniti ma possono essere facilmente implementati utilizzando il costrutto `let`:

  - sorgente:

    ```
    coppiabooleana =W <true, false>;
    fst =W fun x -> let x be <x1,x2> in x1;
    snd =W fun x -> let x be <x1,x2> in x2;
    ```

  - interprete:

    ```
    # coppiabooleana
    : (1 + 1) * (1 + 1) = <inl(<>), inr(<>)>

    # fst coppiabooleana
    : 1 + 1 = inl(<>)
    ```

- **Case-distinction** Il `case f of inl(c) -> g | inr(d) -> h` è un costrutto basato sull'unione disgiunta. Si valuta f finchè non si raggiunge la forma `inl(v)` o `inr(v)`, e a seconda del risultato si restituisce g, effettuando la sostituzione $[v/c]$, oppure h, effettuando la sostituzione $[v/d]$ (c e d devono essere variabili libere che compaiono rispettivamente in g oppure h esattamente una volta):

  - sorgente:

    ```
    case_ex =W fun h -> case h of inl(c) -> fst c
                                | inr(c) -> snd c;
    ```

  - interprete:

    ```
    # case_ex
    : 'a6 * 'a4 + 'a9 * 'a6 -> 'a6 = functional value

    # case_ex inr(coppiabooleana)
    : 1 + 1 = inr(<>)
    ```

- **Loop** Il costrutto `(iter c -> f ) g`, dove c è una variabile libera in f, consente di realizzare un ciclo: g viene sostituita alle occorrenze di

c nel corpo di f, dando origine a una nuova espressione h che viene valutata. Se il risultato è di tipo `inl(k)` la computazione prosegue con la sostituzione $[g/k]$, se il risultato è `inr(j)` si restituisce j. Possiamo utilizzare `iter` per creare una funzione `my_max` che dato un termine in input restituisce il massimo per quel tipo[2]:

```
my_max =W fun start -> (iter z ->
          if succ z = z then inl(z) else inr (succ z)) start;
```

Istanziando ad esempio una variabile `type_3 =W min: 1+(1+1)`, la query `my_max type_3` produrrà lo stesso risultato che otterremmo digitando `succ (succ type_3)`, ossia `inr(inr(<>))`.

- **Unbox** Questo costrutto consente il passaggio da tipi della UC a tipi della WC. Consideriamo l'oggetto della UC `[A] = ({*},A)`, inteso come un *thunk* che viene valutato su richiesta; dato un termine t: [A], `unbox(t)` = A (si veda anche 3.2.2).

3.2.2 Upper Class

I tipi della UC rappresentano le strutture ottenute della costruzione Int, secondo questo schema:

X, Y ::= [A] | X $\otimes$ Y | A.X $\rightarrow$ Y

- **[A]** è un qualsiasi tipo della WC opportunamente impacchettato in un *box*
- **X $\otimes$ Y** è il prodotto cartesiano di due tipi della UC
- **A.X $\rightarrow$ Y** è lo spazio di funzioni indicizzato. Rappresenta il tipo delle funzioni che prendono in input un elemento di X e restituiscono un elemento di Y, utilizzando l'argomento in input al più A volte (dove A

[2] Il codice originale di *max* è tratto da [L10]

è un tipo della WC). Questa precisazione è necessaria per mantenere bassa la complessità in spazio. Se A = 1 è sufficiente scrivere X → Y

I termini della UC sono definiti dalla seguente grammatica (dove le lettere x, y indicano variabili, e s, t termini della UC):

s,t ::= x | $< s, t >$ | let s be $< x, y >$ in t | λx.t | s t | [f] | let s be [c] in t | case f of inl(c) → s | inr(d) → t | copy s as x,y in t | hack(c.f)

- **Costruttore e distruttore del Box** Il costrutto `[f]`, denominato *Box*, consente di utilizzare un termine della WC f dentro un termine della UC, ad esempio:

  – sorgente:

  ```
  true =W inl(<>);     trueU =U [true];
  ```

  – interprete:

  ```
  # trueU
  : 'a0 -> 1 + 1 = functional value

  # trueU <>
  : 1 + 1  = inl(<>)
  ```

  Per l'operazione opposta si utilizza il costrutto `let s be [c] in t`, che riduce s fino a portarlo nella forma $[v]$ e poi sostituisce il valore di v a c, che deve essere una variabile libera in t. Possiamo ad esempio implementare la funzione booleana `and` nella WC e nella UC: `andU` preleva i due parametri e li trasforma in termini della WC, dandoli in pasto alla funzione della WC `andW`; il risultato viene nuovamente trasformato in un termine della UC.

  – sorgente:

```
andW =W fun x -> fun y ->
       if x then (if y then true else false)
       else false;

andU =U fun x -> fun y ->
      let x be [xc] in
         let y be [yc] in [andW xc yc];

(* i due termini seguenti sono equivalenti *)
and_test1 =U andU trueU falseU;
and_test2 =U andU [true] [false];
```

– interprete:

```
# and_test1 <>
: 1 + 1  = inr(<>)     (* false *)
```

- **Coppie** Il costruttore di coppie `<s,t>` è speculare a quello descritto per la WC. Il costrutto `let s be <x,y> in t` consente di estrarre i valori di una coppia: riduce s fino a portarlo nella forma `<v,w>` e poi sostituisce il valore di v e w alle variabili libere x e y, che devono comparire in t una sola volta:

  – sorgente:

```
coppiaU =U <trueU, falseU>;       (*  <inl(<>), inr(<>)>  *)
fstU =U fun x -> let x be <x1,x2> in x1;
test_pair =U fstU coppiaU;
```

  – interprete:

```
# test_pair <>
: 1 + 1  = inl(<>)
```

- **Lambda-astrazione** Per indicare i parametri in input di una funzione usiamo il costrutto, già incontrato negli esempi precedenti, `fun x -> t` dove x è una variabile libera che compare una sola volta in t.

- **Applicazione** `s t` rappresenta l'applicazione di un termine (una funzione) s a un altro termine t; il costrutto può essere generalizzato a n

argomenti. Negli esempi precedenti abbiamo utilizzato questo costrutto per creare le variabili di test.

- **Replicazione degli argomenti** Il costrutto `copy x as x1,x2 in t` valuta x e ne sostituisce il valore a $x1$ e $x2$, variabili libere che devono comparire una sola volta in t. Il costrutto, che può essere iterato n volte, consente di aggirare la limitazione imposta da `fun` e `let`, che richiedono che la variabile x sia presente una sola volta in t. Supponiamo ad esempio di dover usare una variabile più volte dentro al corpo di una espressione:

```
var_multipla =U
     fun x -> copy x as x1,x234 in
               copy x234 in x2, x34 in
                copy x34 in x3, x4 in ... ;
```

- **Case-distinction** `case f of inl(c) -> s | inr(d) -> t` rappresenta una generalizzazione dell'omonimo costrutto definito per la UC. I termini f, $inl(c)$ e $inr(d)$ appartengono alla WC, mentre s e t sono della UC. Tipicamente utilizzeremo il `case` in associazione con `let`:

```
case_example =U
     fun x -> let x be [c] in case c of inl(d) -> s
                                        | inr (d) -> t;
```

- **Hack** Questo operatore permette di costruire tutti quei circuiti non realizzabili con i costrutti predefiniti descrivendo, caso per caso, le azioni che il circuito compie a seconda degli input che riceve (si veda per maggior dettaglio [DLS10a]).

3.2.3 Compilazione da UC a WC

I termini messi a disposizione dalla UC hanno il compito di semplificare l'implementazione dei circuiti a supporto della programmazione bidirezionale: essi potrebbero essere programmati direttamente utilizzando la sola WC, ma in maniera più complicata. La divisione in due classi di termini e tipi ha appunto lo scopo di lasciare al programmatore il compito di realizzare

i circuiti a un livello di astrazione più alto, demandando all'interprete la traduzione automatica da UC a WC.
Per valutare un programma IntML tutti i termini della UC in esso presenti devono essere prima compilati in termini della WC, operazione che viene eseguita da un modulo apposito dell'interprete (per le regole di riduzione rimandiamo a [DLS10a]). Per chiarire questo concetto dichiariamo un termine `zero` nella WC e nella UC:

```
zero_WC =W min: 1+1;        (* zero_WC :W 1 + 1 *)
zero_UC =U [zero_WC];       (* zero_UC :U [1 + 1] *)
```

Come è ovvio aspettarsi, se il termine $t \in$ WC ha tipo A, il corrispondente termine $s \in$ UC creato tramite il costruttore *box* avrà tipo $[A]$. Dopo aver caricato il file contenente le definzioni possiamo valutare i termini:

```
# zero_WC
: 1 + 1 = inl(<>)

# zero_UC
: 'a0 -> 1 + 1 = functional value
```

La valutazione di `zero_WC` dà immediatamente origine al suo valore `inl(<>)`, mentre quella di `zero_UC` produce un valore funzionale. Per valutare `zero_UC` l'interprete lo compila trasformando il suo tipo nel corrispondente tipo della WC, quello di una funzione che produce in output il proprio valore in risposta a un generico input `'a0` (nella fattispecie, il singoletto `<>`). Ogni termine della UC, infatti, è una funzione (un nodo del circuito), che produce i propri output su richiesta. Per ottenere il valore della funzione `zero_UC` dovremo quindi darle l'input atteso:

```
# zero_UC <>
: 1 + 1 = inl(<>)
```

In caso di vere e proprie funzioni della UC, il modulo dell'interprete incaricato della compilazione produce due diversi tipi della WC a seconda che la

funzione sia costante o meno (dove con *costante* intendiamo che ignora il proprio input). Possiamo infatti osservare che la compilazione di un termine $s \in$ UC: `[dom] -> [cod]` dà origine a un termine $s_1 \in$ WC il cui tipo è:

1. `('b * dom) + 1 -> ('b * 1) + cod`
   se s è una funzione che utilizza il proprio input

2. `('b + dom) -> 'c + cod`
   se s è una funzione che ignora il proprio input

La presenza di questi due tipi distinti permette di verificare in maniera semplice se un nodo del circuito restituisce un valore costante; come vedremo concretamente nel caso delle stringhe (5.2.2) questa informazione consente di aumentare l'efficienza della comunicazione tra i nodi del circuito.

3.3 OCaml

OCaml (Objective Caml) è stato creato alla metà degli anni '90 per estendere le funzionalità di Caml, un famoso linguaggio funzionale appartenente alla famiglia di ML. Le sue principali caratteristiche sono:

Controllo dei tipi statico: soltanto programmi correttamente tipati possono essere compilati e quindi eseguiti, il che consente di evitare errori di tipo in fase di esecuzione

Polimorfismo parametrico: una funzione è detta polimorfica se alcuni parametri o il valore di ritorno sono di un tipo che non è necessario specificare. L'analizzatore dei tipi contenuto nel compilatore di OCaml inferisce il tipo più generale per ogni espressione, mentre le variabli sono istanziate al tipo dell'argomento durante l'applicazione della funzione. Un semplice esempio è costituito dalla funzione identità ('a indica un tipo generico):

```
#let id x = x;;
val id : 'a -> 'a = <fun>
```

Una volta applicata la funzione id al suo parametro, tuttavia, essa produrrà un valore dal tipo ben definito: la verifica del tipo è condotta a tempo di compilazione, senza danneggiare l'efficienza del programma al momento dell'esecuzione.

Type inference: l'analizzatore di tipi esaminando un'espressione inferisce il tipo più generale a cui essa appartiene. In caso si debba effettuare un controllo più severo sui tipi, ad esempio sull'input di una data funzione, si può utilizzare una coercizione per ordinare all'analizzatore di associare una espressione ad un tipo più specifico. Tornando all'esempio della funzione identità:

```
# let id (x : int) = x;;
val id : int -> int = <fun>
```

Gestione delle eccezioni: le eccezioni predefinite del linguaggio appartengono a un tipo speciale *exn*, che è possibile estrendere dichiarando nuovi gestori. La definizione avviene inserendo la parola chiave *exception* prima del nome del gestore che, convenzionalmente, deve iniziare con la lettera maiuscola. Sollevare un'eccezione causa la terminazione del programma; è invece possibile calcolare il valore di un'eccezione che, al pari di tutti gli altri tipi del linguaggio, può essere il risultato di una funzione.

***Features* di tipo imperativo:** è possibile effettuare delle modifiche esplicite alla memoria, utilizzando strutture tipicamente imperative come gli array, o i record con campi modificabili (i termini che vogliamo poter cambiare devono essere preceduti dalla parola chiave *mutable*). Ricadono sotto il modello imperativo anche le operazioni di input-output, i loop e le eccezioni. Il programmatore è libero di mescolare stile imperativo e funzionale anche nell'ambito di uno stesso programma.
I tipi primitivi del linguaggio sono interi, booleani, caratteri, stringhe e funzioni, oltre allo speciale tipo *unit* che rappresenta un generico insieme costituito da un solo elemento, con una semantica affine a quella del *void* di C. A partire dai tipi base è possibile costruire coppie o tuple, liste, array, strutture

e unioni.

OCaml è un linguaggio sicuro ed efficiente. La tipizzazione statica garantisce l'assenza di errori di tipo a run-time evitando allo stesso tempo il rallentamento dell'esecuzione da parte di test di tipo dinamici; l'utilizzo del garbage collector dà un ulteriore contributo alla sicurezza e il meccanismo delle eccezioni consente al programma di non trovarsi in uno stato inconsistente dopo l'esecuzione di operazioni illegali. La possibilità di mescolare stile funzionale e imperativo provoca alcune complicazioni a livello implementativo (come la necessità di tenere traccia della rappresentazione della memoria e dell'ordine in cui sono effettuate le operazioni) ma garantisce al programmatore un'estrema flessibilità nella scrittura degli algoritmi.

Capitolo 4

L'interprete per IntML

In questo capitolo richiamiamo pregi e difetti dell'approccio interpretativo rispetto a quello compilativo; descriveremo poi le principali funzionalità messe attualmente a disposizione dall'interprete IntML, fornendone anche una breve guida d'uso.

4.1 Interpreti e Compilatori

I programmi scritti in un qualsiasi linguaggio di programmazione, prima di essere eseguiti, devono essere tradotti in linguaggio macchina, l'unico che la CPU può comprendere direttamente. Fra il linguaggio (generalmente di alto livello) e il linguaggio macchina possono esistere diverse *macchine astratte* intermedie, ognuna delle quali rappresenta un'astrazione del livello sottostante. Nel caso più semplice, immaginiamo di scrivere del codice in un qualsiasi linguaggio di nostra conoscenza, e di avere a disposizione uno strumento (nient'altro che un altro programma) che lo trasforma direttamente in istruzioni per la CPU. Questo strumento può essere un *interprete*, che traduce le frasi del nostro programma mentre le pronunciamo, oppure un *traduttore*, che esegue la traduzione per iscritto e, solo una volta terminata, la consegna alla CPU. Veniamo ora ad una definizione più formale.

Usiamo il pedice L per indicare che un particolare costrutto si riferisce al linguaggio L, e l'apice Lo per indicare che un programma è scritto nel linguaggio Lo. Un generico programma P realizza una funzione parziale $\mathcal{P}^L$: $\mathcal{D} \to \mathcal{D}$ tale che $\mathcal{P}^L$(Input) = Output se l'esecuzione di $\mathcal{P}^L$ su Input termina, oppure è indefinita se l'esecuzione di $\mathcal{P}^L$ su Input non termina. $\mathcal{D}$ rappresenta un arbitrario insieme di dati.

- un *interprete* è un programma, che indicheremo con $\mathcal{I}_L^{Lo}$, scritto in linguaggio Lo e che interpreta tutte le istruzioni scritte in L. L'interprete realizza una funzione parziale:

  $\mathcal{I}_L^{Lo}$: $(\mathcal{A}^L$ x $\mathcal{D}$) $\to \mathcal{D}$ t.c. $\mathcal{I}_L^{Lo}$ $(\mathcal{P}^L$, Input) = $\mathcal{P}^L$(Input)

  dove $\mathcal{A}^L$ denota l'insieme di tutti i possibili programmi scritti in linguaggio L. L'interprete effettua una traduzione implicita: sostituisce ogni istruzione di L con una serie di istruzioni in Lo, ma senza produrre in uscita del codice. La fase di traduzione non è distinta da quella di esecuzione.

- un *compilatore* è un programma, che indicheremo con $\mathcal{C}_{L,Lo}$, che prende in input un un programma $\mathcal{P}^L$ scritto in L e lo traduce in un programma compilato $\mathcal{PC}^{Lo}$ scritto in Lo, che andrà poi eseguito con input $\mathcal{D}$. Un compilatore realizza una funzione (non necessariamente parziale):

  $\mathcal{C}_{L,Lo}$: $\mathcal{A}^L \to \mathcal{A}^{Lo}$ t.c. se $\mathcal{C}_{L,Lo}(\mathcal{A}^L) = \mathcal{PC}^{Lo}$, allora $\mathcal{P}^L(\mathcal{D}) = \mathcal{PC}^{Lo}(\mathcal{D})$

  Il compilatore effettua una traduzione esplicita delle istruzioni scritte in L, producendo in ouput un nuovo programma scritto in Lo.

Implementazione interpretativa e compilativa presentano vantaggi e svantaggi:

- *efficienza*: ad ogni istruzione l'interprete deve decodificare i costrutti del linguaggio di partenza, effettuando una nuova decodifica per ogni occorrenza dello stesso comando, mentre il compilatore esegue la decodifica di una istruzione una volta per tutte, anche in caso di più occorrenze. La fase di compilazione può essere lunga ma viene eseguita una

volta sola, mentre con l'interpretazione occorre sommare al tempo di esecuzione delle istruzioni il tempo necessario alla loro interpretazione;

- *occupazione di memoria*: sebbene al giorno d'oggi questa non sia in genere una grossa limitazione, l'utilizzo di un interprete non richiede la produzione di nuovo codice e dunque riduce l'occupazione di memoria;

- *flessibilità*: l'utilizzo di un interprete facilta l'interazione col linguaggio sorgente, il che agevola tra l'altro la creazione di strumenti di debugging e di tracing. La compilazione invece causa la perdita di informazioni riguardo alla struttura del programma sorgente, e potrebbe essere più difficile determinare il comando che ha realmente generato un errore a run-time;

- *facilità di programmazione*: in generale è più semplice progettare un interprete che un compilatore e dunque si preferiscono soluzioni interpretative quando si vuole implementare un linguaggio in tempi relativamente brevi.

4.2 L'interprete per IntML

All'avvio, l'interprete richiede come parametro il percorso di un file sorgente IntML (estensione *.iml*); dopo il caricamento del file sarà possibile valutare le definizioni in esso presenti. L'interprete è in grado di valutare le espressioni contenute nel file sorgente o quelle predefinite del linguaggio ma non consente di creare nuove associazioni nome-valore a tempo di esecuzione.

4.2.1 Regole per la valutazione dei termini

Per valutare una espressione della WC è sufficiente digitare il suo nome dopo il prompt, mentre per le espressioni della UC il nome deve essere seguito da <> (si veda 3.2.3). L'utente ha inoltre a disposizione una serie di keywords, introdotte da `#`, per richiedere alcuni comportamenti specifici, ad esempio:

- `#instancetype` *type*: in caso di termini contenenti variabili libere, impone all'interprete di utilizzare il tipo specifico indicato
- `#eval` *query*: è l'opzione di default, equivale a digitare semplicemente *query*

Nel capitolo 5 ci occuperemo estesamente dell'implementazione di una nuova keyword, `#explore`.

- sorgente:

```
zero =W min: 1+(1+(1+1));
zeroU =U [zero];
```

- interprete

```
# zero                (* come #eval zero *)
: 1 + (1 + (1 + 1)) = inl(<>)

# zeroU               (* come #eval zeroU *)
: 'a0 -> 1 + (1 + (1 + 1)) = functional value

# zeroU <>
: 1 + (1 + (1 + 1)) = inl(<>)
```

Capitolo 5

Esplorazione di Dati Strutturati

L'implementazione di funzioni tramite i costrutti dell'UC di IntML consente la progettazione di circuiti i cui nodi - rappresentati dalle funzioni stesse - instaurano una comunicazione bidirezionale, come già illustrato al cap. 3 e come vedremo più in concreto analizzando il caso delle stringhe. Al momento non esistono veri programmi interamente scritti in IntML ma sono state prodotte alcune librerie, tra cui quella delle funzioni aritmetiche in $\mathbb{LOGSPACE}$ realizzata da Michael Lodi ([L10]), e alcune utility come l'algoritmo di `Selection Sort` in $\mathbb{LOGSPACE}$ di Federico Foschini [F10]. I termini IntML che abbiamo fin qui creato e riportato per illustrare i costrutti di questo linguaggio sono estremamente semplici e possono essere valutati immediatamente dall'interprete. Lavorando invece su tipi di dato strutturati, quali stringhe, alberi o grafi, il programmatore non ha a disposizione efficaci strumenti di debugging, il che rende la fase di progettazione e di testing degli algoritmi più faticosa del dovuto. Sembra opportuno arricchire l'interprete con una funzionalità, che chiameremo `#explore`, grazie alla quale l'utente può visualizzare immediatamente a schermo il contenuto di certi tipi di dato strutturati. In questo capitolo ci occupiamo in particolare dell'esplorazione di stringhe e alberi binari, definendo con precisione questi tipi di dato e la loro

semantica, e illustrando in dettaglio il codice che implementa la nuova *feature* dell'interprete. Creeremo anche una funzione IntML `inverti_albero` che, dato un albero binario booleano in input, restituisce l'albero coi nodi di segno opposto.

5.1 Migliorare l'Interazione con l'Interprete

Immaginiamo di voler progettare una funzione `inverti` che, presa in input una stringa, restituisca la stringa inversa. Intuitivamente, dovendo lavorare in spazio logaritmico, `inverti` dovrà accettare la stringa in input e produrre quella in output un bit alla volta, e la stringa in input sarà anch'essa una funzione, che per ogni input $i : 1...n$ restituisce il valore dell'i-esimo bit. Abbiamo quindi creato un circuito, che potrà avere altri nodi come ad esempio una terza funzione, o l'utente da linea di comando, che interroga `inverti`. Immaginiamo ora di avere implementato `inverti` e di volerla testare: dobbiamo invocarla su n distinti input e prendere nota del risultato a parte! Il risultato di `inverti` applicato a `input_string` infatti è una stringa, e una stringa è una funzione, motivo per cui l'interprete non può valutarla restituendo un valore:

- file sorgente:

```
input_string =U fun x ->  ....  (* input_string = 000111 *)
inverti =U fun ...
...
mytest =U inverti input_string;
```

- interprete:

```
# mytest
'a -> 1 + 1 = functional value
```

A fronte di questo problema, vogliamo arricchire l'interprete con una nuova keyword `#explore` che, applicata a termini e tipi conformi a certe strutture prestabilite, aiuti il programmatore in fase di testing, arrivando a un risultato di questo tipo:

```
# #explore mytest
: string
= 111000
```

In questo capitolo presenteremo la funzionalità di esplorazione applicata a due diversi tipi di dato: le stringhe e un modello elementare di albero binario con nodi che possono assumere i valori 0/1. Al di là delle differenze fra questi due tipi, l'approccio più intuitivo prevede una serie di passi elementari che può essere così riassunto:

1. si controlla se la query inserita è strutturalmente compatibile con una funzione che rappresenti una stringa o un albero
2. in caso affermativo, si analizza il dominio della funzione per creare una lista di tutti i possibili input
3. si richiama la funzione su tutti gli input, producendo un insieme di tutti gli output che viene poi aggregato e stampato a schermo.

5.2 Stringhe

Nell'ambito di un linguaggio funzionale puro come IntML una stringa lunga n bit può essere convenientemente rappresentata da una funzione che prende in input un intero compreso tra 1 e n (rappresentante la posizione del bit richiesto), e che restituisce il valore dell'iesimo bit (0, 1 o l'eventuale *bottom* a indicare la fine della stringa). Le stringhe sono dunque funzioni con tipo riconducibile a:

1. s: [n] $\rightarrow$ [2]
2. t: [n] $\rightarrow$ [3]

dove non siamo interessati al tipo concreto di dominio e codominio - che non possiamo vincolare a modelli prestabiliti - bensì alla loro cardinalità.
Implementare una funzionalità di esplorazione delle stringhe a livello dell'interprete significa fornire un modello di elaborazione di queste funzioni volutamente generico e privo di qualsiasi interpretazione semantica. Sarà

compito del programmatore invocare `#explore` su una funzione che rappresenta effettivamente una stringa, e il cui risultato visualizzato sullo schermo abbia un qualche significato, a lui noto e utile. D'altra parte, un utente che voglia progettare una funzione che rappresenta una stringa deve rispettare un unico vincolo riguardante il codominio, quale che sia il suo tipo effettivo: **i valori del codominio devono essere ordinati, ossia il primo valore deve rappresentare lo 0, il secondo 1 e l'eventuale terzo il *bottom*.**

5.2.1 Esempi di Stringhe

Riportiamo di seguito alcuni esempi di stringhe che potremmo visualizzare a schermo tramite la keyword `#explore`.
La prima funzione, `w_string_min`, ha una lunghezza che dipende dal tipo del parametro x. La stringa vale `0010...` se $[x] \geq 4$, `00` se $[x] = 2$ e via dicendo. Il codominio ha esattamente due valori, il dominio contiene una variabile libera: per visualizzare questa stringa dovremo prima istanziarne il dominio tramite la keyword `#instancetype`, scegliendo un tipo la cui cardinalità corrisponda alla lunghezza che vogliamo ottenere.

- sorgente:

```
(* valori di ritorno - solo per chiarezza *)
zero =W min: 1+1;                (* inl(<>) *)
uno =W succ zero;                (* inr(<>) *)

w_string_min =W fun x ->
      if (x = min) then zero
         else if (x = (succ min)) then zero
              else if (x = (succ(succ min))) then uno
                   else zero;
```

- interprete:

```
# w_string_min
: 'a10 -> 1 + 1 = functional value
This query term contains free type variables in annotations.
For evaluation they are instantiated to 1.
```

La seconda funzione che vediamo, `w_string_bin`, è come `w_string_min` ma con dominio istanziato al tipo del termine `start0`, con cardinalità 7. Ne risulta che `w_string_bin` = 0010000. Ridefinendo `start0` nel file sorgente modificheremo il tipo del dominio della stringa e, di conseguenza, la sua lunghezza. Possiamo poi facilmente implementare la stessa funzione nell'UC:

- sorgente:

```
start0 =W min: 1+(1+(1+((1+1)*(1+1))));

(* stringa 0010000 *)
w_string_bin =W fun x ->
       if (x = start0 ) then zero
           else if (x = (succ start0) ) then zero
                else if (x = ( succ (succ start0))) then uno
                     else zero;

u_string_bin =U fun x -> let x be [c] in [w_string_bin c];
bintest =U u_string_bin [(succ (succ start0))];
```

- interprete:

```
# w_string_bin
: 1 + (1 + (1 + (1 + 1) * (1 + 1))) -> 1 + 1 = functional value

# u_string_bin
: 1 * (1 + (1 + (1 + (1 + 1) * (1 + 1)))) + 1
                         -> 1 * 1 + (1 + 1)  = functional value

# w_string_bin (succ (succ start0))
: 1 + 1 = inr(<>)

# bintest <>
: 1 + 1 = inr(<>)
```

Per conoscere il valore di `w_string_bin` dobbiamo richiedere la valutazione dell'interprete su tutti i possibili input e tenerne traccia manualmente; nel caso di `u_string_bin` dobbiamo anche dichiarare nel file sorgente un termine di test per ogni possibile input (come `bintest` nell'esempio) e poi valutarlo

da linea di comando. Per quanto riguarda le funzioni con codominio a tre valori, consideriamo la stringa `w_string_ter`, lunga *al massimo* 4 (pari alla cardinalità del tipo del parametro `start`) ma che in effetti è lunga solo 2:

```
start =W min: 1+(1+(1+1));

(* 3 possibili valori ritorno *)
zero =W min: (1*1)+(1*1)+(1*1);   uno =W succ zero;   end =W succ uno;

(* stringa 11 *)
w_string_ter =W fun x ->
         if (x = start) then uno
             else if (x = (succ start)) then uno else end;
```

Consideriamo infine una funzione di ordine superirore, `addL`, che prende in input due parole binarie e restituisce in output la loro somma[1]:

```
addL :U (['a] --> [1 + 1]) --> (['a] --> [1 + 1]) --> ['a] --o [1 + 1]
```

La funzione `addL` realizza un circuito di comunicazione bidirezionale che può essere così descritto: per ottenere il bit i della stringa in output occorre lanciare `addL` con input i; la funzione chiederà alle stringhe $w1$ e $w2$ non solo il bit i ma anche tutti quelli che lo precedono, poichè la computazione è *stateless* (per motivi di spazio non è possibile memorizzare i risultati intermedi) ed è necessario conoscere il riporto generato dall'addizione dei bit $i - 1$. Per ottenere l'output completo di `addL` occorre includere nel file sorgente una serie di definizioni di questo genere, una per ogni bit delle stringhe di partenza:

```
testadd1 =U addL w1 w2 ([(min)]: ['a] );          (* testadd1 :U [1 + 1] *)
testadd2 =U addL w1 w2 ([(succ min)]: ['a] ]);    (* testadd2 :U [1 + 1] *)
...
```

Una volta lanciato l'interprete potremo valutare uno ad uno i termini di test, segnare a parte il loro valore (0 oppure 1) e ricostruire così la stringa. Siamo ovviamente liberi di definire un termine `result` a rappresentare la stringa in output, ma non abbiamo alcun modo di valutarlo nella sua globalità.

[1] Per l'implementazione si veda [L10].

- sorgente:

```
result =U addL w1 w1;
testresult1 =U result ([min]: ['a]);    (* primo bit dell'output *)
```

- interprete:

```
#result
: 1 * ('a0 + ('a1 + 'a2)) + 1 -> 1 * 1 + (1 + 1) = functional value

# testresult1 <>
: 1 + 1 = inr(<>)
```

La keyword `#explore` che vogliamo implementare ci consentirà di superare questo ostacolo, velocizzando in maniera significativa le fasi di testing e di debugging.

5.2.2 Stringhe della WC e UC a Confronto

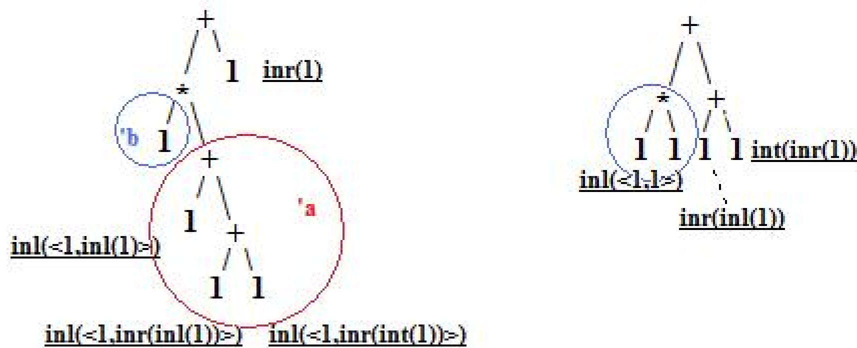
La struttura delle stringhe della UC fornisce l'occasione per tornare sulla differenza tra i termini della UC e i corrispondenti termini della WC, fornendo nuovi dettagli sull'implementazione dei circuiti che stanno alla base della programmazione bidirezionale (3.2.3). Partiamo dalla definzione di una semplice stringa `001` nella WC, definendo poi il termine corrispondente nella UC:

```
zero =W min: 1+1;
uno =W succ zero;

stringa_WC =W fun x ->          (* stringa_WC :W 1 + (1 + 1) -> 1 + 1 *)
     if (x = min: 1+(1+1)) then zero
        else if ( x = (succ min: 1+(1+1))) then zero
             else uno;

stringa_UC =U fun y ->     (* stringa_UC :U [1 + (1 + 1)] --o [1 + 1] *)
     let y be [c] in [stringa_WC c];
```

La valutazione dell'interprete sarà:

```
# stringa_WC
: 1 + (1 + 1) -> 1 + 1 = functional value

# stringa_UC
: 1 * (1 + (1 + 1)) + 1 -> 1 * 1 + (1 + 1) = functional value
```

Il tipo compilato di `stringa_UC` presenta la seguente struttura, dove per maggiore leggibilità abbiamo sostituito il singoletto `<>` con `1`:



Dominio: 1* (1 + (1 + 1)) + 1 **Codominio: 1 * 1 + (1 + 1)**

La compilazione traduce il tipo della UC di `stringa_UC` nel tipo corrispondente della WC: tale passaggio, eseguito automaticamente dall'interprete, nasconde al programmatore i dettagli dell'implementazione delle reti di message passing. La struttura del tipo compilato di `stringa_UC` necessita di qualche chiarimento. Possiamo fare la seguente osservazione: **la compilazione di un termine funzionale $s \in$ UC**: `['a] --o [ 1 + 1 ]` **produce un termine $s_1 \in$ WC**: `('b * 'a) + 1 --> ('b * 1) + ( 1 + 1 )` **ad esso equivalente**.

`stringa_WC` è una normale funzione unidirezionale che prende in input un indice e restituisce il valore del bit che si trova in quella posizione, mentre `stringa_UC` è un nodo di message-passing (collocabile dentro una rete di arbitraria grandezza) ossia una funzione che realizza una comunicazione bidirezionale, col seguente funzionamento:

1. la comunicazione con la funzione viene avviata lanciandole come primo input `inr(<>)`;

2. se la funzione può restituire immediatamente un valore, poichè ignora di fatto il proprio input, lo produce sotto forma di `inr(inl(<>))` oppure `inr(inr(<>))`. Calcolando la lunghezza della stringa a partire dal tipo `'a` del dominio si potrà ricostruire il suo contenuto senza interrogarla ulteriormente.

3. se la funzione utilizza il proprio input, in risposta a `inr(<>)` genera in output il valore `inl(<<>,<>>)`; il nodo chiamante può così iniziare a inviarle gli input `inl(<<>, inl(<>)>)`, `inl(<<>, inr(inl(<>))>)` e `inl(<<>, inr(inr(<>))>)`, che rappresentano la posizione del bit di cui vuole conoscere il valore.

Supponiamo ora di voler produrre una stringa i cui bit valgono tutti 0, che con un lieve abuso di linguaggio definiremo stringa *costante*. Per consentire al compilatore di riconoscere una stringa di questo tipo occorre usare un piccolo accorgimento nella sua definizione:

```
constant_string_WC =W                            (* 'a -> 1 +1 *)
        fun x -> min: 1+1;

invalid_constant_UC =U             (* ['a0] --o [1 + 1] *)
        fun x -> let x be [c] in [constant_string_WC c];

valid_constant_UC1 =U                  (* 'b --> [1 + 1] *)
        fun x -> [constant_string_WC (min: 1+(1+1))];

valid_constant_UC2 =U                  (* 'b --> [1 + 1] *)
        fun x -> [min: 1+(1+1)];
```

`invalid_constant_UC` è correttamente definita ma non può essere interpretata come stringa costante poichè nel suo corpo si fa uso del parametro `x`, e il compilatore non può riconoscere che la chiamata a `constant_string_WC` produce un valore costante; `valid_constant_UC1` e `valid_constant_UC2`

sono banalmente costanti in quanto ignorano il proprio parametro. La differenza tra la stringa costante non valida e quelle valide è puramente sintattica, e si riflette nel diverso tipo del loro dominio, che soltanto nel caso di `invalid_constant_UC` è - coerentemente - un tipo della UC. Per ricostruire il contenuto di `invalid_constant_UC` dovremo interrogarla su tutti gli input, sebbene essa ci restituisca in output sempre 0, ma per ricostruire le stringhe contrassegnate come `valid` è sufficiente una sola interrogazione.
Ripetiamo ancora che la traduzione da termine della UC a termine della WC è realizzata automaticamente dall'interprete: il programmatore dovrà occuparsi soltanto dell'implementazione delle funzioni della UC senza curarsi dei dettagli relativi all'implementazione della comunicazione bidirezionale tra i nodi di message-passing.

5.2.3 #explore *string*

`#explore` deve essere in grado di interpretare come stringa ogni funzione che, come detto, appartenga ai tipi [n] → [2] oppure [n] → [3], e deve poter essere applicata tanto ai termini della WC che a quelli della UC. Intuitivamente dovremo effettuare i seguenti passaggi:

1. analizzare il tipo di *string*, che deve essere un termine funzionale;
2. analizzare il tipo del codominio del termine e calcolarne la cardinalità, che deve essere necessariamente pari a 2 oppure a 3;
3. analizzare il tipo del del dominio e generare una lista di tutti i possibili valori in input;
4. applicare il termine *string* ad ogni valore di input, valutare il risultato, mapparlo associandolo ai valori `0/1/bottom` e stamparlo a schermo.

Se *string* fallisce uno dei test di tipo, la computazione si interrompe e l'interprete si mette in attesa di una nuova richiesta, stampando un messaggio di errore. Vediamo ora in dettaglio l'implementazione di `#explore` applicata a un parametro di tipo stringa.

Main `string_main` riassume tutte le operazioni necessarie alla valutazione di una stringa. I suoi parametri sono `ds`, le dichiarazioni contenute nel file sorgente arricchite con le annotazioni di tipo, la `query` che deve essere esplorata, e la variabile `inst_value` in cui abbiamo memorizzato il tipo da associare alle variabili libere (modificabile dall'utente con la keyoword `#instancetype`). Per nostra comodità, mentre risaliamo al tipo della query cercandola dentro le dichiarazioni del file, utilizziamo la flag booleana `wc` per ricordare se la query è un termine della WC o della UC. Dopo aver verificato che la query sia effettivamente un termine funzionale procediamo con la sua compilazione, che consiste nel legare tutte le variabili e nel trasformare l'eventuale tipo della UC in tipo della WC. Verifichiamo poi se la query può rappresentare una stringa e, in caso affermativo, generiamo la lista degli input e degli output, che passeremo alle funzioni di valutazione `eval_string_WC` e `eval_string_UC`, a seconda del tipo. Il risultato della valutazione è espresso sotto forma di stringa, che stampiamo a schermo prima di restituire il controllo all'interfaccia interattiva dell'interprete.

In caso di stringhe costanti (ossia che ignorano il proprio parametro, come abbiamo mostrato al paragrafo precedente) dobbiamo usare un piccolo accorgimento: a una stringa costante della UC di tipo `'a0 -> [cod]` corrisponde un tipo compilato `'a0 + 'a1 -> 'a2 + (cod)`, in cui compaiono ben tre variabili libere, di cui una corrisponde al vero e proprio dominio e due sono generate dalla compilazione. La funzione `create_type_list` (descritta più sotto) sostituisce ad ogni variabile libera il tipo memorizzato in `temp_instance_type`: ciò significa che una stringa *costante* di tipo ('a0) $\rightarrow$ [1 + 1], dove ad esempio 'a0 = 1 + 1, una volta compilata ha tipo: ('a0 + 'a1) $\rightarrow$ 'a2 + (1 + 1) (si vedano 5.2.2 e 3.2.3), che `create_type_list` interpreta come ((1 + 1) + (1 + 1)) $\rightarrow$ (1 + 1) + (1 + 1) dando origine a liste di input e output scorrette. Per evitare problemi di questo genere, nel `main` salviamo la cardinalità del dominio effettivo della funzione UC in una variabile globale `constant_dom`, che utilizzeremo all'atto della valutazione, in caso di stringhe costanti, per ricostruirne correttamente il contenuto.

```
1  let string_main (ds: typed_decls)(query: Term.t)(inst_value: Type.t): unit=
2   temp_instance_type := inst_value;
3   constant_dom := 0;
4   let query_type = find_and_print_type ds query in
5     (match query_type with
6     | Type.FunA(_,_) -> wc := true;
7     | Type.FunB (_,_,_) -> wc := false;
8     | _ -> print_error "function");
9
10  (*compilazione della query*)
11  let ground_query, closed_query_type = produce_compiled_query query ds
12  in     (*controllo che sia una stringa*)
13    if (is_a_string closed_query_type) then
14      begin
15      match closed_query_type with
16      (*creo la lista degli input e degli output*)
17      | Type.FunA (dom, cod) ->
18        let listOfVal = create_type_list dom in
19          if (!wc = false) then
20              match dom with
21              | Type.SumA(t1,t2) -> constant_dom := cardinalita t1;
22              | _ -> ();
23          else ();
24          let listOfOutput = create_type_list cod in
25            if (!wc) then (eval_string_WC ground_query listOfVal
                  listOfOutput)
26            else (eval_string_UC ground_query listOfVal listOfOutput);
27      | _ -> failwith "error: string_main"
28      end
29     else print_error "stringa";;
```

Controlli sul tipo La query viene sottoposta a due tipi di controllo. Come prima cosa verifichiamo che si tratti di un termine funzionale, ovvero che sia in forma di `FunA` oppure `FunB` (che rappresentano nel codice dell'interprete il tipo delle funzioni per WC e UC). Superato questo test, verifichiamo che la funzione possa rappresentare una stringa calcolandone la cardinalità del codominio. La funzione booleana `is_a_string` cerca di individuare uno dei due modelli attesi: in particolare, le stringhe della WC hanno un tipo `[dom] -> [cod]` t.c. `cod` $\subseteq \{2, 3\}$, mentre quelle (già compilate) della UC hanno tipo `[1 * dom] + 1 -> [(1 * 1) + cod]` t.c. `cod` $\subseteq \{2, 3\}$ (si veda

5.2.2). La flag booleana `wc` limita il rischio di scambiare per stringa quello che potrebbe rappresentare altro, e viceversa, e semplifica la struttura del *pattern matching*.

```
1  (* stringa WC = FunA (dom,cod), calcoliamo la cardinalita' di cod *)
2  (* stringa UC = FunA(dom, cod) con cod = SumA(TensorA(_,OneA), truecod)),
3  calcoliamo la cardinalita' di truecod *)
4  let is_a_string (tipo: Type.t): bool =
5    match tipo with
6    | Type.FunA (dom, cod) ->
7      if (!wc) then          (* WC query *)
8        (let codominio = cardinalita cod in
9          if ((codominio = 2) || (codominio = 3)) then true else false;)
10     else                   (* UC query *)
11     (match cod with
12     | Type.SumA (fst, snd) ->
13       (* fst = (1*'b) se e' una stringa normale, = 'a0 se e' una stringa
           costante *)
14         let codominio = cardinalita snd in
15           if ( (codominio = 2) || (codominio = 3) ) then true else false;
16     | _ -> false; (* il cod non e' un Sum *) )
17   | _ -> failwith "error: is_a_string";;
```

La funzione che calcola la cardinalità prende in considerazione soltanto tipi finiti del prim'ordine (le eventuali funzioni *higher-order* della UC a questo punto dell'esecuzione sono state già compilate nei tipi della WC: gli ultimi due casi del match sono ridondanti). In caso di variabili di tipo calcoliamo la cardinalità del valore memorizzato nella variabile `temp_instance_type`.

```
1  let cardinalita (tipo: Type.t): int =
2     let rec card (ty: Type.t) = match ty with
3       | Type.SumA (t1, t2) -> (card t1) + (card t2)
4       | Type.TensorA (t1, t2) -> (card t1 ) * (card t2)
5       | Type.Var(a) -> card !temp_instance_type;
6       | Type.ZeroA | Type.OneA -> 1
7       | Type.FunA (_,_) -> failwith "no cardinality for FunA!"
8       | _ -> failwith "no cardinality for UC terms!";
9     in card tipo;;
```

Liste di Input/Output La funzione `create_type_list` crea ricorsivamente, a partire da un tipo passato come parametro, la lista di tutti i termini che lo compongono. Ci è sembrato ragionevole imporre un tipo di ordinamento per cui `inl(t)` viene prima di `inr(t)` e `fst(pair)` viene prima di `snd(pair)`.

```
1  let rec create_type_list (ty: Type.t)  =
2   let rec makeleft list  = match list with
3     | [] -> []
4     | head :: tail -> (Term.mkInlA head) :: makeleft tail
5   and makeright list = match list with
6     | [] -> []
7     | head :: tail -> (Term.mkInrA head ):: makeright (tail)
8   and makepair l1 l2 =
9   (*prodotto tra un termine e una lista di termini*)
10    let rec makeaux term l2 = match l2 with
11      | [] -> []
12      | head:: tail -> Term.mkPairA term head :: makeaux term tail
13      in
14      match l1 with
15        | [] -> []
16        | head:: tail -> (makeaux head l2) @ (makepair tail l2)
17  in
18  match ty with
19  | Type.OneA -> [Term.mkUnitA]
20  | Type.SumA (t1, t2) ->
21    (makeleft (create_type_list t1)) @ (makeright (create_type_list t2))
22  | Type.TensorA (t1, t2) ->
23    makepair (create_type_list t1) (create_type_list t2)
24  | Type.Var(a) -> create_type_list !temp_instance_type;
25  | _ -> failwith "create_type list: UC types not implemented";;
```

Valutazione dei termini della WC La funzione `eval_string_WC` esegue tutte le operazioni necessarie per la valutazione delle stringhe della WC. Al suo interno, `evalW` compone ricorsivamente la funzione con tutti i termini della lista degli input, passandoli a `valuta` che ne calcola il valore; la funzione `map_out` cerca l'output nella lista degli output, restituendone la posizione (0 per il bit 0, 1 per il bit 1, e l'eventuale 2 a indicare fine della stringa). Per interrompere forzatamente la ricorsione in caso di funzioni con codominio a

tre valori viene sollevata l'eccezione `End_of_string`.

```
let eval_string_WC (query: Term.t) (lista_input: Term.t list)
                              (lista_output: Term.t list): string =
 let buf = Buffer.create 80 in
   let map_out (output: Term.t) (lista: Term.t list): int =
   begin
     let rec search_term termine ll pos =
     match ll with
     | head :: tail -> if (termine = head) then pos
                       else (search_term termine tail (pos+1));
     | [] -> failwith "error: map_out"
     in
     let posizione = search_term output lista 0 in posizione
   end
   in
     let valuta (termine: Term.t): unit =
       let valutazione = (Evaluation.eval_closed termine) in
         match valutazione  with
         | Some v -> let pos =
           map_out v lista_output in
           if (pos = 2) then      (*stringa finita*)
           (Buffer.add_string buf "(e)"; raise End_of_string)
           else
           (Buffer.add_string buf (string_of_int pos);
           Buffer.add_string buf " ");
         | None -> failwith "Unexpected None in valuta";
     in
       let rec evalW (qu: Term.t)(li: Term.t list)(lo: Term.t list) =
       match li with
       | [] -> (); (* NOP, finita valutazione *)
       | head :: tail -> valuta (Term.mkAppA qu head); evalW qu tail lo;
 in
   (try evalW query lista_input lista_output with End_of_string -> ());
 Buffer.contents buf;;
```

Valutazione dei termini della UC Come già spiegato in 5.2.2, all'atto di valutare una stringa della UC la funzione chiamante - in questo caso la nostra funzione di valutazione - instaura con essa una comunicazione bidirezionale. Il primo input inviato alla stringa è il termine `inr(<>)`, a cui essa risponderà con `inl(t,t)` a indicare di proseguire la valutazione, o con valori `inr(t)` a indicare che si tratta di una funzione costante.

La funzione `eval_string_UC` ha una struttura analoga a quella progettata per la WC, ma con un piccolo accorgimento: all'inizio della valutazione viene invocata una speciale funzione `save_first_value`, che interroga la stringa sull'input `inr(<>)` e salva la risposta nella variabile `constant_value`, coi seguenti valori: 0 per il bit 0, 1 per il bit 1, l'eventuale 2 per fine della stringa, 3 per il valore `inl(t)` che, ripetiamo, può essere restituito solo nella prima interrogazione. Se la stringa da valutare non è costante, ossia utilizza in modo effettivo il suo parametro, `evalRecUC` invoca ricorsivamente `valuta` sulla lista dei suoi input, altrimenti lancia `build_constant_string` che semplicemente stampa n volte il valore memorizzato in `constant_value`, dove n è la cardinalità del dominio della stringa (che, nel `main`, salviamo nella variabile globale `constant_dom`).

```
1  let eval_string_UC (query: Term.t) (lista_input: Term.t list)
      (lista_output: Term.t list): string =
2    let buf = Buffer.create 80
3    in
4    let map_UC_out (output: Term.t) (lista: Term.t list): int =
5      (let rec search_term termine ll pos =
6        match ll with
7        | head :: tail ->
8          (match head.desc with
9          | Term.InlA(t) -> search_term termine tail pos;
10         | Term.InrA (t) -> if (termine = head) then pos
11                            else (search_term termine tail (pos+1));
12         | _ -> failwith "map_UC_out: error";)
13       | [] -> failwith "output not found in map_UC_out";
14     in
15     match output.desc with
16     | Term.InlA(t) -> 3; (* la stringa non e' costante.*)
17     | _ -> let posizione = search_term output lista 0 in posizione;)
18   in
19   (* interroga query con inr(<>), salva l'output 0/1/2/3 nella var globale
       constant_value  *)
20   let save_first_value (query: Term.t) (lista_output: Term.t list): unit =
21     (let first_output_term =
22       Evaluation.eval_closed (Term.mkAppA query
           (Term.mkInrA(Term.mkUnitA))) in
23         match first_output_term with
24         | Some v -> let pos = (map_UC_out v lista_output) in
             (constant_value := pos);
```

```
25          | None -> failwith "error: is_constant_string";)
26   in
27   let valuta (termine: Term.t): unit =
28     let valutazione = (Evaluation.eval_closed termine) in
29       match valutazione  with
30       | Some v  -> let output = (map_UC_out v lista_output) in
31                    if (output = 2) then (Buffer.add_string buf "(e)";
32                                          raise End_of_string)
33                    else (Buffer.add_string buf (string_of_int output);
34                          Buffer.add_string buf " ");
35       | None -> failwith "Error: valuta(evalUC)";
36   in
37   let rec build_constant_string (acc: int): unit =
38     (match acc with
39     | 0 -> Buffer.add_string buf "(constant)";
40     | _ -> Buffer.add_string buf (string_of_int !constant_value);
41            Buffer.add_string buf " "; build_constant_string (acc-1);)
42   in
43   let rec evalRecUC (qu: Term.t) (li: Term.t list) (lo: Term.t list): unit =
44     match li with
45     | [] -> (); (* non ci arriviamo mai *)
46     | head :: tail when (head = (Term.mkInrA(Term.mkUnitA))) -> ();
47                  (* inr(<>) era il primo val che abbiamo dato in input *)
48     | head :: tail -> if (!constant_value != 3)  (* stringa costante *)
49                       then  build_constant_string !constant_dom
50                       else                 (* stringa non costante  *)
51                       (valuta (Term.mkAppA qu head); evalRecUC qu tail lo);
52   in
53   let () = save_first_value query lista_output in
54     (try evalRecUC query lista_input lista_output with End_of_string -> ());
```

5.2.4 Test

Vediamo il comportamento di `#explore` su alcune tipologie di stringhe, per ognuna delle quali presentiamo un termine della WC e uno della UC. Nell'esempio sulle stringhe *higher-order* (stringhe come risultato di funzioni) utilizziamo la funzione `addL` [L10] già descritta in 5.2.1.

Stringa [n] → [2]

```
w_string_bin =W fun x ->                (* stringa 0010 *)
     if (x = start0 ) then zero
       else if (x = (succ start0) ) then zero
```

```
          else if (x = ( succ (succ start0))) then uno
             else zero;

u_string_bin =U fun x -> let x be [c] in [w_string_bin c];

(* interprete *)
# #explore w_string_bin            # #explore u_string_bin
: string                           : string
= 0 0 1 0                          = 0 0 1 0
```

Stringa [n] → [3]

```
w_string_ter =W fun x ->                 (* stringa 11 *)
    if (x = newstart) then due2
        else if (x = (succ newstart)) then due2
            else tre2;
u_string_ter =U fun x -> let x be [c] in [w_string_ter c]

(* interprete *)
# #explore w_string_ter             # #explore u_string_ter
: string                            : string
= 1 1 (e)                           = 1 1 (e)
```

Stringa Vuota

```
emptystring =W fun x ->
     if (x = zero) then  succ (succ (min: 1+(1+1)))
         else succ (succ (min: 1+(1+1)));
emptyUCstring =U fun x -> let x be [c] in [emptystring c];

(* interprete *)
# #explore emptyUCstring
: string
= (e)
```

Stringa Costante

```
constant_string_WC =W fun x -> min: 1+1;
constant_string_UC =U fun x -> [constant_string_WC (min: 1+(1+1))];
```

```
(* interprete *)
# constant_string_UC
: 'a0 + 'a1 -> 'a2 + (1 + 1) = functional value

# #explore constant_string_UC
: string
= 0 (constant)

(* modifichiamo il tipo da associare ad 'a0: *)
# #instancetype 1+(1+(1+1))

# #explore constant_string_UC
: string
= 0 0 0 0 (constant)
```

Stringa *higher-order*

```
w1 =U ....        (* w1 :U ['a + ('b + 'c)] --o [1 + 'd]  = 111 *)
w2 =U .....       (* w2 :U ['a + ('b + 'c)] --o [1 + 1]   = 110 *)
addL :U (['a] --> [1 + 1]) --> (['a] --> [1 + 1]) --> ['a] --o [1 + 1]
provaw1w2 =U addL w1 w2;

(* interprete *)
# #explore provaw1w2
: string
= 0 1 0
```

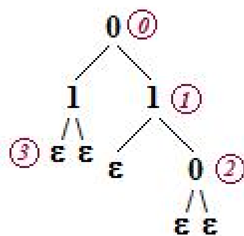
5.3 Alberi Binari

Vogliamo implementare il tipo di dato albero binario in IntML, limitandoci per semplicità a un tipo elementare i cui nodi possono assumere soltanto i valori 0/1. Occorre trovare un modo conveniente di esprimere l'albero tramite un termine funzionale che permetta di scrivere algoritmi $\subseteq \mathbb{LOGSPACE}$, il che - come è facile intuire - significa rinunciare alla ricorsività della definizione tradizionale: un albero composto da n nodi, infatti, richiede una occupazione

di memoria almeno lineare rispetto a n. Immaginiamo che ai nodi dell'albero siano associate delle etichette numeriche: interrogando la funzione che rappresenta l'albero sul valore dell'i-esimo nodo non è possibile dare un limite all'occupazione di memoria dell'output, che avrebbe dimensioni diverse a seconda della profondità del nodo stesso (foglia, radice o nodo intermedio). Non potendo dare una definzione ricorsiva dell'albero, ne utilizzeremo una lineare:

```
treeRec n = | Empty;              treeIntML n = | Empty;
            | 0 (t1, t2);                       | 0 (n1, n2);
            | 1 (t1, t2);                       | 1 (n1, n2);
```

La funzione ricorsiva `treeRec` prende in input un intero n e restituisce tutto il sottoalbero radicato nel nodo passato in input (`t1` e `t1` sono alberi), mentre `treeIntML` restituisce il valore del nodo n e l'etichetta dei suoi figli (`n1` e `n2` sono interi). Vediamo come si presentano in OCaml le due funzioni che descrivono l'albero in figura (i numeri nel cerchietto rappresentano l'etichetta del nodo):




```
(*    ---------------------    definizioni    ---------------------    *)
(*    ---- ricorsivo ----   *)              (*    ---- lineare ----    *)
type rectree =                            type lintree =
  | EmptyR                                  | Empty
  | ZeroR of rectree * rectree              | ZeroL of int * int
  | OneR of rectree * rectree;;             | OneL of int * int;;

let rec treeRec (n: int): rectree =       let treeIntML (n: int): lintree =
  match n with                              match n with
  | 0 -> ZeroR(treeRec 3, treeRec 1);       | 0 -> ZeroL (3, 1);
```

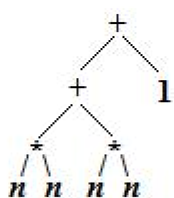
```
  | 1 -> OneR(EmptyR, treeRec 2);          | 1 -> OneL (4, 2);
  | 2 -> ZeroR (EmptyR, EmptyR);           | 2 -> ZeroL (4, 4);
  | 3 -> OneR (EmptyR, EmptyR);            | 3 -> OneL (4, 4);
  | _ -> failwith "Non presente";;         | 4 -> EmptyL;
                                           | _ -> failwith "Non presente";;

(*    ---------------------    valutazione    ---------------------    *)
# treeRec 1;;                            # treeIntML 1;;
- : rectree = OneR (EmptyR,              - : lintree = OneL (4, 2)
              ZeroR (EmptyR, EmptyR))

# treeRec 0;;                             # treeIntML 0;;
- : rectree =                             - : lintree = ZeroL (3, 1)
ZeroR (OneR (EmptyR, EmptyR),
       OneR (EmptyR, ZeroR (EmptyR, EmptyR)))
```

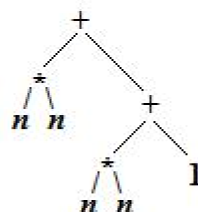
La dimensione degli output di `treeRec` dipende dalla profondità del nodo in input, quella degli output di `treeIntML` è fissa e calcolabile a priori in funzione dell'occupazione di spazio dell'intero n. Il tipo di albero binario a valori 0/1 può essere quindi rappresentato in IntML da funzioni che prendono in input un intero n e restituiscono una tripla costruita dal valore del nodo e dall'etichetta dei suoi due figli. Più precisamente, gli alberi sono funzioni con tipo riconducibile a:

1. s: [n] $\rightarrow$ (([n] * [n]) + ([n] * [n])) + 1

2. t: [n] $\rightarrow$ ([n] * [n]) + (([n] * [n]) + 1)

in cui [n] è una qualsiasi composizione finita di tipi interpretabile come un intero, e i due tipi distinti di codominio hanno lo scopo di lasciare maggiore libertà al programmatore. Vediamo la struttura dei due codomini, con le relative funzioni di mappatura degli output:





0(a, b) = inl(inl(<a, b>))
1(a, b) = inl(inr(<a, b>))
empty = inr(1)

0(a, b) = inl(<a, b>)
1(a, b) = inr(inl(<a, b>))
empty = inr(inr(1))

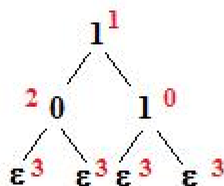
Per ricostruire la struttura di un albero IntML occorre prima invocare su tutti i possibili input la funzione che lo rappresenta, e poi procedere con la sostituzione dei valori dei nodi. A differenza delle stringhe, in cui l'ordine di emissione degli output è progressivo (e consente di ricostruire la stringa in maniera lineare), per gli alberi occorre identificare la radice e incorporarle man mano i nodi dei livelli inferiori. Questa operazione, se si procede per tentativi, può diventare onerosa in termini di tempo di calcolo, e se fatta a mano si rivela proibitiva anche per alberi di dimensione ridotta (si veda anche 5.3.3). Incorporare nell'interprete una funzionalità di esplorazione automatica degli alberi consentirà invece al programmatore di visualizzare immediatamente a schermo qualsiasi tipo di albero binario implementato seguendo le regole che abbiamo definito. Vogliamo quindi progettare una funzione `#explore` che trasformi un albero binario lineare (ossia, una foresta di nodi) in un albero binario ricorsivo, per arrivare a un risultato di questo tipo:

```
# treeIntML
: 'a -> 'a * 'a + 'a * 'a + 1 = functional value

# #explore treeIntML
: tree
= 0( 1(empty, empty), 1(empty, 0(empty, empty)))
```

5.3.1 Esempi di Alberi

Il semplice albero binario rappresentato in figura può essere descritto in diversi modi. In questo esempio definiamo due funzioni della WC `funtree1`

(albero di tipo 1) e `funtree2` (albero di tipo 2) e le corrispondenti funzioni della UC `funtree1U` e `funtree2U`.

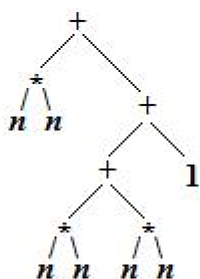


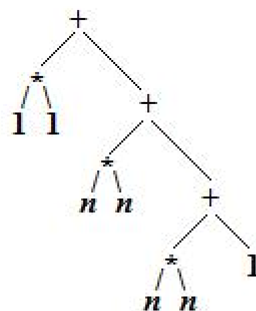

```
(*   ----------------------   sorgente   ---------------------   *)
zero =W min: 1+(1+(1+1));
uno =W succ zero;
due =W succ uno;
tre =W succ due;

(* (([n]*[n]) + ([n]*[n])) + 1 *)
funtree1 =W fun x ->                  (* (([n]*[n]) + ([n]*[n])) + 1 *)
  if (x = zero) then inl(inr(<tre,tre>))               (* 0 = 1(3,3) *)
    else ( if (x = uno) then inl(inr(<due,zero>))      (* 1 = 1(2,0) *)
      else ( if (x = due) then inl(inl(<tre,tre>))     (* 2 = 0(3,3) *)
        else inr (<>)));                                (* 3 = empty *)

funtree2 =W fun x ->                  (* ([n]*[n]) + (([n]*[n]) + 1) *)
  if (x = zero) then inr(inl(<tre,tre>))               (* 0 = 1(3,3) *)
    else ( if (x = uno) then inr(inl(<due,zero>))      (* 1 = 1(2,0) *)
      else ( if (x = due ) then inl(<tre,tre>)         (* 2 = 0(3,3) *)
        else inr(inr(<>))));                            (* 3 = empty *)

funtree1U =U fun x -> let x be [c] in [funtree1 c];
funtree2U =U fun x -> let x be [c] in [funtree2 c];

(*    ----------------------  interprete   -----------------------   *)
# funtree1
: 1 + (1 + (1 + 1)) ->
   (1 + (1 + (1 + 1))) * (1 + (1 + (1 + 1))) +
     (1 + (1 + (1 + 1))) * (1 + (1 + (1 + 1))) + 1
= functional value
```

```
# funtree2U
: 1 * (1 + (1 + (1 + 1))) + 1 -> 1 * 1 +
   ((1 + (1 + (1 + 1))) * (1 + (1 + (1 + 1))) +
     ((1 + (1 + (1 + 1))) * (1 + (1 + (1 + 1))) + 1))
= functional value
```

5.3.2 Alberi della UC e WC a confronto

Fra gli alberi della WC e quelli della UC sussiste la medesima differenza di tipo che abbiamo rilevato per le stringhe (5.2.2): **la compilazione di un termine funzionale** $s \in$ **UC**: [dom] `--o` [cod] **produce un termine** $s_1 \in$ **WC**: ('b * dom) + 1 → ('b * 1) + cod **ad esso equivalente**. Nel caso degli alberi del primo tipo, ad esempio, ciò significa che il tipo della funzione della UC ['a] `--o` [(('a * 'a) + ('a * 'a)) + 1] sarà compilato nel tipo corrispondente ('b * 'a) + 1 → ('b * 1) + ((('a * 'a) + ('a * 'a)) + 1), dove 'b vale 1. Vediamo la struttura dei due possibili codomini di alberi della UC, con le relative funzioni di mappatura degli output:





0(a, b) = inr(inl(inl(<a, b>)))
1(a, b) = inr(inl(inr(<a, b>)))
empty = inr(inr(1))

0(a, b) = inr(inl(<a, b>))
1(a, b) = inr(inl(inl(<a, b>)))
empty = inr(inr(inr(1)))

5.3.3 Alberi e Grafi: Topsort

Il primo passo per la ricostruzione di un albero IntML consiste, come vedremo in dettaglio nella sezione successiva, nel valutare la funzione su tutti i possibili input generando una lista di valori che, per comodità, saranno costituiti da coppie `<int, lintree>` dove il primo elemento rappresenta l'etichetta del nodo e il secondo il suo valore. Ad esempio, la valutazione dell'albero `funtree1` presentato in 5.3.1 genera la seguente lista: [(0, 1(3,3)); (1, 1(2,0)); (2, 0(3,3)); (3, empty)], dove i nodi non sono ordinati per profondità ma in base alle loro etichette, che dipendono dalla struttura interna della funzione. Per ricostruire l'albero conviene procedere rintracciando la radice e, di seguito, i nodi che da essa dipendono ma, se n è la lunghezza della lista, la ricerca di ogni singolo nodo costa n nel caso pessimo. Per risolvere questo problema, e allo stesso tempo verificare se la funzione IntML rappresenta un vero albero o una struttura degenere (nulla vieta che ci siano cicli, o più radici), possiamo utilizzare il `Topological Sort` o `Topsort`, un algoritmo che ordina i vertici di un grafo aciclico in modo che, per ogni arco *(u,v)* del grafo, il nodo u precede il nodo v nell'ordinamento. Un albero binario, infatti, non è altro che un particolare tipo di grafo orientato in cui un nodo è designato come radice e ogni nodo - tranne la radice - ha esattamente due archi uscenti e uno entrante. Il funzionamento dell'algoritmo è semplice, e consente un ordinamento nel grafo in $\mathcal{O}(|V| + |E|)$, ossia lineare nel numero di nodi e archi. Abbiamo apportato qualche modifica alla versione originale di `Topsort` per adattarlo meglio alla struttura che stiamo considerando; l'algoritmo risultante può essere così descritto:

1. partendo dalla lista generata dalla funzione di valutazione dell'albero, si crea una lista `R` contenente tutte le radici, restituendo errore se la lista è vuota (c'è un ciclo diretto) o se contiene più di un elemento, altrimenti si procede;

2. ad ogni iterazione si preleva un elemento da `R`: se il valore del nodo è `empty` lo si inserisce in una lista `F`, se è `0(_,_)` o `1(_,_)` lo si inserisce in `S`, che conterrà la soluzione ordinata, e si inseriscono in coda a `R` i suoi due figli, prima il sinistro e poi il destro;

3. quando R è vuoto l'algoritmo è finito: appendiamo F in coda a S e restituiamo in output la lista così ordinata.

La particolare codifica che abbiamo dato dell'albero rende putroppo l'algoritmo abbastanza oneroso: la stessa ricerca delle radici, ad esempio, ha tempo quadratico in n. Rispetto a un vero grafo, inoltre, occorre tener conto del numero dei nodi *empty*, che per alberi di una certa profondità può diventare considerevole. Per gestire tali nodi utilizziamo la lista F non presente nell'algoritmo originario, in cui inseriamo i nodi con valore *empty* controllando però che non siano già presenti; questa lieve complicazione è necessaria poichè nulla vieta, in linea di principio, che nodi distinti siano *empty* (negli esempi finora considerati ce n'era soltanto uno), e devono essere presi tutti in considerazione per ricostruire correttamente l'albero. D'altra parte, la lista ordinata per livelli consente di ricostruire l'albero in tempo ottimale poichè, per produrre il sottoalbero radicato nel nodo in posizione i, occorrerà scandire soltanto gli elementi $(i{+}1)...n$ della lista. Continuando il nostro esempio, la valutazione *ordinata* di `funtree1` genera la seguente lista: [(1, 1(2,0)); (2, 0(3,3)); (0, 1(3,3)); (3, empty)] e la ricostruzione dell'albero ora è immediata. Di seguito riportiamo il codice completo della nostra implementazione di `Toposort`:

```
1 (* prende in input la descrizione dell'albero (lista di coppie int *
     lintree) e restituisce la lista ordinata *)
2 let topsort (lista: (int * lintree) list): (int * lintree) list =
3
4   (* ---- utility per manipolazione liste ---- *)
5   let enqueue elem lista = match lista with
6   | [] -> [elem];
7   | head :: tail -> lista @ [elem];
8   in
9   (*inserisco in coda sse non presente*)
10  let rec enqueue_empty elem lista = match lista with
11  | [] -> [elem];
12  | head :: tail -> if (head = elem) then lista else enqueue_empty elem
       tail;
13  in
14  let rec remove elem lista = match lista with
15  | [] -> []
```

```
16  | head :: [] when (head != elem) -> failwith "remove: element not found ";
17  | head :: tail -> if (elem = head) then tail
18       else (if ((List.hd tail) = elem) then head :: (List.tl tail)
19         else remove elem tail);
20  in
21  let rec search_node (n: int) (mylist: (int * lintree) list): (int *
        lintree) =
22  match mylist with
23  | [] -> failwith "search element: not found";
24  | head:: tail -> if ( (fst head) = n) then head else search_node n tail;
25  in
26  (*scandisce la lista dei nodi del grafo tree_desc, produce in output la
        lista delle radici*)
27  let rec findradix tree_desc elenconodi: (int * lintree) list =
28     let rec israd (n: int) (lista: (int * lintree) list) : bool =
29     match lista with
30     | [] -> true;
31     | head :: tail ->
32  (match (snd head) with
33  | ZeroL (num1, num2) -> if ( (n = num1) || (n = num2)) then false
34       else israd n tail;
35  | OneL (num1, num2) -> if ( (n = num1) || (n = num2)) then false
36       else israd n tail;
37  | EmptyL -> israd n tail;)
38     in
39     match tree_desc with
40     | [] ->  [];
41     | head :: tail -> if (israd (fst head) elenconodi) then (head ::
          findradix tail elenconodi)
42      else findradix tail elenconodi;
43  in
44 (*ciclo principale dell'algoritmo di sorting. produce in output la lista
       ordinata*)
45  let rec toploop (rad_list: (int * lintree) list) sol flush start: (int *
        lintree) list =
46    match (List.length rad_list) with
47    | 0 -> let sol = sol @ flush in sol;   (*finito*)
48    | _ -> begin
49     let nodo = (List.hd rad_list) in
50    (*prelevo il primo elemento della lista radici*)
51    ( let rad_list = remove nodo rad_list in
52    (*controllo il valore del nodo rimosso*)
53    match (snd nodo) with
54    | EmptyL -> let flush = enqueue_empty nodo flush in
55          toploop rad_list sol flush start;
56    | ZeroL (num1, num2) ->
57      (*inserisco i figli in rad_list e il nodo nella sol*)
```

```
58        let rad_list = enqueue (search_node num1 start) rad_list  in
59        let rad_list = enqueue (search_node num2 start) rad_list in
60        let sol = enqueue nodo sol in
61        toploop rad_list sol flush start;
62      | OneL (num1, num2) ->
63        (*inserisco i figli in rad_list e il nodo nella sol*)
64        let rad_list = enqueue (search_node num1 start) rad_list in
65        let rad_list = enqueue (search_node num2 start) rad_list in
66        let sol =  enqueue nodo sol in
67        toploop rad_list sol flush start;)
68      end;
69    in
70    let radix_queue = findradix lista lista in
71    if ((List.length radix_queue) > 1) then failwith "topsort: found more
          than 1 radix"
72    else if ((List.length radix_queue) = 0) then failwith "topsort, direct
          cycle detected"
73    else let sol = toploop radix_queue [] [] lista in sol;;
```

5.3.4 #explore *tree*

Per ricostruire un albero binario con `#explore` effettueremo i seguenti passaggi:

1. verificare che *tree* sia un termine funzionale;
2. analizzare il tipo di *tree* verificando che sia compatibile con uno dei tipi prestabiliti;
3. generare la lista di tutti i possibili input;
4. valutare *tree* su tutti gli input creando una lista di coppie (*input* * *output*)
5. mappare la lista in base al tipo di albero in input, trasformandola in un tipo omogeneo di lista di coppie `<int`, `linetree>` (si veda pag. 56);
6. ordinare la lista in modo che la radice sia in prima posizione e le foglie in fondo;
7. ricostruire ricorsivamente l'albero, passando dalla lista di coppie `<int`, `linetree>` a un unico termine ricorsivo di tipo `rectree`, e stamparlo a schermo.

La computazione viene interrotta se *tree* fallisce uno dei test iniziali di tipo oppure se, dopo la valutazione, l'algoritmo di ordinamento dei nodi riscontra la presenza di cicli all'interno dell'albero o l'esistenza di più radici.

Main I parametri di `tree_main` sono `ds`, le dichiarazioni contenute all'interno del file sorgente, la `query` che deve essere esplorata e la variabile `inst_value` in cui abbiamo memorizzato il tipo da associare alle eventuali variabili libere presenti nel termine. Si cerca la query dentro all'elenco delle definizioni, accedendo al suo tipo; si procede con la compilazione della funzione e con la generazione della lista dei suoi input. Si lancia la funzione di valutazione - distinta a seconda che il termine sia della WC o della UC - e si stampa a schermo il contenuto dell'albero.

```
1  let tree_main (ds: typed_decls) (query: Term.t) (inst_value: Type.t): unit =
2
3    temp_instance_type := inst_value;
4    tree_type := 0;
5
6    let query_type = find_and_print_type ds query in
7      (match query_type with
8      | Type.FunA(_,_) -> wc := true;
9      | Type.FunB (_,_,_) -> wc := false;
10     | _ -> print_error "function");
11
12   let ground_query, closed_query_type = produce_compiled_query query ds in
13     if (is_a_tree closed_query_type) then
14         (match closed_query_type with
15         | Type.FunA(t1,t2) ->
16             (let listOfInput = create_type_list t1 in
17               if (!wc)  then Printf.printf"\n: %s\n" (eval_tree_WC
                   ground_query listOfInput)
18         else Printf.printf"\n: %s\n" (eval_tree_UC ground_query listOfInput))
19         | _ -> failwith "tree main";)
20     else failwith "not a tree";;
```

Controlli sul tipo I controlli da effettuare sul tipo di dato albero sono più onerosi di quelli visti in precedenza per le stringhe, dove ci limitavamo a calcolare la cardinalità del codominio. Se la verifica ha esito positivo, salviamo nella variabile `tree_type` i valori 1 o 2 a seconda della struttura del codominio dell'albero. Tale informazione è necessaria per una corretta mappatura degli output della funzione.

```
(* verifica se il tipo TY rappresenta un albero; in caso affermativo salva
    nella var. globale tree_type il suo tipo (1 oppure 2) *)
let is_a_tree (ty: Type.t) : bool =

  (* rest. true se i tipi FIRST e SND  sono uguali *)
  let rec compare (first: Type.t) (snd: Type.t): bool =
    match first with
    | Type.Var(_) -> (match snd with
                  | Type.Var(_) -> true;
                        | _ -> false;)
    | Type.OneA -> if (snd = Type.OneA) then true else false;
    | Type.ZeroA -> if (snd = Type.ZeroA) then true else false;
    | Type.TensorA (t1,t2) ->
          (match snd with
          | Type.TensorA (t3,t4) -> ((compare t1 t3) && (compare t2 t4));
          | _ -> false;)
    | Type.SumA (t1,t2) ->
          (match snd with
          | Type.SumA (t3, t4) ->  ((compare t1 t3) && (compare t2 t4));
          | _ -> false;)
    | _ -> failwith "higher-order compare not implemented";
  in
  match ty with
  | Type.FunA (dom, cod) ->

    if (!wc) then     (* WC *)

      (let prod = Type.TensorA (dom, dom)  (* prod = n*n *) in
  let proof1 = Type.SumA(Type.SumA(prod,prod), Type.OneA) in
          if (compare cod proof1) then (tree_type :=1; true)
          else
          (let proof2 = Type.SumA(prod, Type.SumA(prod, Type.OneA)) in
      if (compare cod proof2) then (tree_type :=2; true) else false; ))

    else    (* UC *)

    (match dom with       (* dom UC = (1 * domWC) + 1 *)
    | Type.SumA (fst, _) ->
      (match fst with
      | Type.TensorA(_, realdom) -> let prod = Type.TensorA (realdom,
          realdom) in
        (match cod with
        | Type.SumA (t1, t2) ->
           let proof1 = Type.SumA(Type.SumA(prod,prod), Type.OneA) in
             (if (compare proof1 t2) then (tree_type :=1; true)
               else (let proof2 = Type.SumA(prod, Type.SumA(prod,
                   Type.OneA)) in
```

```
45                 (if (compare proof2 t2) then (tree_type :=2; true) else
                       false; )))
46         | _ -> false;)
47       | _ -> failwith "is_a_tree";)
48     | _ -> failwith "";)
49  | _ -> failwith "";;
```

Valutazione dei termini della WC La funzione `eval_tree_WC` esegue la valutazione di `query` sulla lista degli input `in_list` e restituisce la descrizione dell'albero in forma di stringa. In particolare, `evalWT` applica ricorsivamente il termine che rappresenta l'albero alla lista degli input; la funzione `valuta` esegue la valutazione dell'output e produce una coppia di termini che, mappata da `map_pair`, viene trasformata in coppia `int * lintree`. La valutazione produce dunque una lista di coppie input-output, che viene prima passata a `topsort` per essere ordinata (si veda 5.3.3) e poi a `build_tree` per la ricostruzione dell'albero.

```
1  let eval_tree_WC (query: Term.t) (in_list: Term.t list): string =
2
3   (* mappa NUMERO nell'intero corrispondente, cercandolo nella lista degli
       input *)
4   let mappa (numero: Term.t): int =
5     let rec map_number (numb: Term.t) (acc: int) (li: Term.t list): int =
6       match li with
7       | [] -> failwith "map_number: input not found";
8       | head :: tail -> if (head.desc = numb.desc) then acc else (map_number
            numb  (acc+1) tail);
9     in map_number numero 0 in_list
10  in
11  (* trasforma la coppia in input in coppia int * lintree *)
12  let map_pair (coppia: Term.t * Term.t): (int * lintree) =
13    let first = mappa (fst coppia) in
14      let second =
15        if (!tree_type = 1) then            (* albero tipo 1 *)
16        begin
17        match (snd coppia).desc with
18        | Term.InrA (_) -> EmptyL;
19        | Term.InlA (t1) ->
20            (match t1.desc with
21            | Term.InlA(pair) ->
22              (match pair.desc with
```

```
23             | Term.PairA (c1,c2) -> ZeroL(mappa c1, mappa c2);
24             | _ -> failwith "map_pair: pair expected";)
25           | Term.InrA(pair) ->
26             (match pair.desc with
27             | Term.PairA (c1,c2) -> OneL(mappa c1, mappa c2);
28             | _ -> failwith "pair expected in map_pair";)
29           | _ -> failwith "map_pair: type mismatch";)
30       | _ -> failwith "map_pair: output type mismatch 1";
31       end
32       else if (!tree_type = 2) then        (* albero tipo 2 *)
33       begin
34       match (snd coppia).desc with
35       | Term.InlA (pair) ->
36           (match pair.desc with
37           | Term.PairA (c1, c2) -> ZeroL (mappa c1, mappa c2);
38           | _ -> failwith "pair expected in map_pair";)
39       | Term.InrA (t) ->
40           (match t.desc with      (* inl(<n,n>) oppure inr(<>) *)
41           | Term.InrA(_) -> EmptyL;
42           | Term.InlA (pair) ->
43               (match pair.desc with
44               | Term.PairA (c1,c2) -> OneL(mappa c1, mappa c2);
45               | _ -> failwith "pair expected in map_pair";)
46           | _ -> failwith "map_pair: output type mismatch 2";)
47       | _ -> failwith "output type mismatch 3 in map_pair";
48       end
49       else failwith "unknow tree type in map_pair";
50 in (first, second)
51 in
52 (* restituisce la coppia (input * output) mappata in (int * lintree) *)
53 let valuta (funct: Term.t) (input: Term.t) =
54   let valutazione = Evaluation.eval_closed (Term.mkAppA funct input) in
55   match valutazione with
56   | Some v ->  map_pair (input, v);
57   | None -> failwith "valuta: None inside evaluation";
58 in
59 (* produce la lista completa di coppie input-output*)
60 let rec evalWT (qu: Term.t) (li: Term.t list): (int * lintree) list  =
61 match li with
62 | [] -> [];
63 | head :: tail -> (valuta qu head) ::  evalWT qu tail;
64 in
65 let output_list = evalWT query in_list in
66   let sorted_output = topsort output_list in
67     let result = build_tree sorted_output in (print_tree result);;
```

Valutazione dei termini della UC La funzione di valutazione degli alberi della UC è concettualmente identica a quella della WC; l'unica differenza è che il primo input dato all'albero della UC è `inr(<>)`, ossia l'ultimo valore della lista degli input (per la spiegazione si riveda la valutazione delle stringhe UC in 5.2.2). Se la prima risposta dell'albero è `inl(<<>,<>>)` la computazione procede normalmente, altrimenti si arresta.

```
1
2 let eval_tree_UC (query: Term.t) (in_list: Term.t list): string  =
3
4 (* trasforma la coppia (Term.t Term.t) in  (int * lintree) *)
5 let map_output_UC (coppia: Term.t * Term.t) (lis: Term.t list): int *
    lintree  =
6  let map_n_UC (numb: Term.t) (l: Term.t list): int =
7     let rec mappainput (n: Term.t) (ly: Term.t list) (acc: int): int =
8     match ly with
9     | [] -> failwith "map_n_UC: input not found";
10    | head :: tail ->
11      (match head.desc with
12   (* o sono inl di coppie oppure e' l'ultimo termine, inr (<>) *)
13   | Term.InrA(_) -> failwith "map_n_UC: found inr(<>)";
14   | Term.InlA (pair) ->
15     (match pair.desc with
16     | Term.PairA(t1,t2) -> if t2.desc = n.desc then acc else mappainput n
         tail (acc+1);
17    | _ -> failwith "mappainput";)
18   | _ -> failwith "map_n_UC: term";)
19     in mappainput numb l 0;
20   in
21   (* mappa in numeri termini del dominio *)
22   let map_inputUC (inp: Term.t) (ly: Term.t list): int =
23   let rec mi (i: Term.t) (li: Term.t list) (acc: int): int =
24   match li with
25   | [] -> failwith "map_inputUC: input not found";
26   | head :: tail ->  if (head.desc = i.desc) then acc else mi i tail (acc +
       1);
27   in mi inp ly 0;
28   in
29   let first = map_inputUC (fst coppia) lis in
30     let second =
31     begin
32     if (!tree_type = 1) then
33       begin
34       match (snd coppia).desc with
```

```
35        | Term.InrA(t1) ->
36          (match t1.desc with
37          | Term.InrA(_) -> EmptyL;
38          | Term.InlA(t2) ->
39            (match t2.desc with
40            | Term.InlA (pair) ->
41              (match pair.desc with
42              | Term.PairA(n1,n2) -> ZeroL((map_n_UC n1 lis), (map_n_UC n2
                    lis));
43              | _ -> failwith "mapUc 1";)
44            | Term.InrA(pair) ->
45              (match pair.desc with
46              | Term.PairA(n1,n2) -> OneL((map_n_UC n1 lis), (map_n_UC n2
                    lis));
47              | _ -> failwith "mapUc t";)
48            | _ -> failwith "mapUC 2";)
49          | _ -> failwith "mapUC 3";)
50     (* | Term.InlA(pair) -> impossibile *)
51        | _ -> failwith "pamUC 4";
52        end
53      else if (!tree_type = 2) then
54      (match (snd coppia).desc with
55      (* | Term.InlA(pair) -> Printf.printf " inl(<1,1>)\n" : impossibile *)
56      | Term.InrA(t1) ->
57        (match t1.desc with
58        | Term.InlA(pair) ->
59          (match pair.desc with
60          | Term.PairA (n1,n2) -> ZeroL((map_n_UC n1 lis), (map_n_UC n2 lis));
61          | _ -> failwith "mapUC 5";)
62          | Term.InrA (term) ->
63            (match term.desc with
64            | Term.InlA(pair) ->
65              (match pair.desc with
66              | Term.PairA (n1,n2) -> OneL((map_n_UC n1 lis), (map_n_UC n2
                    lis));
67              | _ -> failwith "mapUC 6";)
68            | Term.InrA(_) -> EmptyL;
69            | _ -> failwith "mapUC 5";)
70          | _ -> failwith "mapUC 7";)
71      | _ -> failwith "mapUC 8";)
72      else failwith "unknown UC tree type"
73      end
74    in (first, second);
75    in
76    let valutaU (funct: Term.t) (input: Term.t) (*: int * lintree *) =
77      let valutazione = Evaluation.eval_closed (Term.mkAppA funct input) in
78      match valutazione with
```

```
79      | Some v -> map_output_UC (input, v) in_list;
80      | None -> failwith "valuta: None inside evaluation";
81   in
82   (* produce una lista di coppie input—output di tipo int * lintree.
83   L'input inr(<>)  e' valutato per primo tramite first_query*)
84   let rec evalUT (qu: Term.t) (li: Term.t list): (int * lintree) list =
85   match li with
86   | [] -> [];
87   | head :: tail when (head = (Term.mkInrA(Term.mkUnitA))) -> [];
88   | head :: tail -> (valutaU qu head) ::  evalUT qu tail;
89   in
90   (* rest. false se tree non ha risposto inl(<*,*>) *)
91   let first_query (tree: Term.t): bool =
92     let reply = Evaluation.eval_closed (Term.mkAppA tree
         (Term.mkInrA(Term.mkUnitA))) in
93       (match reply with
94       | Some v -> (match v.desc with
95                    (* inl significa continua, se inr = valore costante *)
96                    | Term.InlA(_) ->  true;
97                    | Term.InrA(_) -> false;
98                    | _ -> failwith "first_query: reply";)
99       | None -> failwith "first_query: none";)
100  in if (first_query query) then
101    (let output_list = evalUT query in_list in
102      let sorted_output = topsort output_list in
103        let result = build_tree sorted_output in (print_tree result);)
104     else failwith "tree gave a constant response: not implemented";;
```

Ricostruzione dell'albero Una volta ottenuta la lista ordinata (per l'algoritmo di ordinamento si veda 5.3.3) la ricostruzione dell'albero è immediata. Partendo dalla radice si procede scandendo la lista in avanti e sostituendo progressivamente i termini di tipo `lintree`: si ottiene così un unico termine ricorsivo `rectree` che, passato alla funzione `print_tree`, viene poi restituito al `main` sotto forma di stringa.

```
1 let build_tree (lista: (int * lintree) list): rectree =
2   (* prende in input un lintree NODE e la lista dei termini da NODE
        compreso in avanti *)
3   let rec subst (node: lintree) (li: (int * lintree) list) =
4     let rec search_element (n: int) (mylist: (int * lintree) list) =
5     match mylist with
6     | [] -> failwith "search_element: not found";
```

```
7     | head:: tail -> if ( (fst head) = n) then (snd head) else
          search_element n tail;
8   in
9   match node with
10  | EmptyL -> EmptyR;
11  | ZeroL (n1, n2) -> (* la lista diminuisce ad ogni passo *)
12     (match li with
13     | [] -> failwith "it can't happen";
14     | head :: tail ->
15       ZeroR(subst (search_element n1 tail) tail, subst (search_element n2
             tail) tail);)
16  | OneL (n1, n2) ->
17     (match li with
18     | [] -> failwith "it can't happen";
19     | head :: tail ->
20       OneR(subst (search_element n1 tail) tail, subst (search_element n2
             tail) tail);)
21  in
22  match lista with
23  | [] -> EmptyR;
24  | head :: tail -> subst (snd head) lista;;
```

5.3.5 Test

Effettuando il test sugli alberi d'esempio presentati in 5.3.1 otteniamo il seguente risultato:

```
# #explore funtree1
: tree
= 1(0(empty,empty),1(empty,empty))

# #explore funtree1U
: tree
= 1(0(empty,empty),1(empty,empty))
```

Vogliamo ora scrivere una semplice funzione IntML `inverti_albero` che, dato un albero 0/1 di tipo 1, restituisce l'albero i cui nodi hanno valore invertito. La funzione prende in input un albero binario `tree` e un indice `index`, chiede a `tree` il valore del nodo `index` e produce in output il nodo con valore opposto rispetto a quello di partenza.

```
(* inverti_albero :U (['b] --> ['b * 'b + 'b * 'b + 1]) --o
                              ['b] --> ['b * 'b + 'b * 'b + 1] *)
inverti_albero =U
   fun tree: {'b}['a] --o [('a * 'a + 'a * 'a) + 1] ->
      fun index: ['a] ->
          let (tree index) be [result] in
              (case result of
               inl(level1) -> (case level1 of
                                inl(level2) -> [inl(inr(level2))]
                                | inr(level2) -> [inl(inl(level2))])
             | inr(level1) -> [inr(level1)]);
```

Le annotazioni di tipo che abbiamo applicato ai parametri della funzione hanno il mero compito di aiutare il programmatore a capire se l'albero in input è di tipo 1 o 2; senza di esse, infatti, l'interprete inferirebbe un tipo generico (['a] `->` ['c + 'd + 'e]) `-o` ['a] `->` ['d + 'c + 'e] che, sebbene corretto, risulta ambiguo.
Proviamo ora ad applicare `inverti_albero` all'albero di esempio `funtree1U`: per testare l'otuput dovremmo invocare la funzione su tutti i possibili valori del dominio e agglomerare manualmente il risultato ma, grazie a `#explore`, possiamo semplicemente testare il risultato finale:

- sorgente:

```
funtree1U =U ....
inverti_albero =U ...

(* rest. il nodo con etichetta 0 *)
prova =U inverti_albero funtree1U [min: 'a];

(* output completo *)
reversetree =U inverti_albero funtree1U;
```

- interprete:

```
# prova <>
: (1 + (1 + (1 + 1))) * (1 + (1 + (1 + 1)))
             + (1 + (1 + (1 + 1))) * (1 + (1 + (1 + 1))) + 1
```

```
= inl(inl(<inr(inr(inr(<>))), inr(inr(inr(<>)))>))

# reversetree
: 1 * 1 * (1 + (1 + (1 + 1))) + 1 ->
        1 * 1 * 1 + ((1 + (1 + (1 + 1))) * (1 + (1 + (1 + 1))) +
         (1 + (1 + (1 + 1))) * (1 + (1 + (1 + 1))) + 1)
= functional value

# #explore funtree1U
: tree
= 1(0(empty,empty),1(empty,empty))

# #explore reversetree
: tree
= 0(1(empty,empty),0(empty,empty))
```

5.4 Codice Completo di #explore

Riportiamo di seguito il codice completo dell'implementazione di `#explore`, con la sola eccezione della funzione `produce_compiled_query`, invocata nel `main`, che fa parte del nucleo originario del codice dell'interpete.

```
1  open Typing
2  open Decls
3  open Printing
4  open Term
5  open Evaluation
6  open Compile
7
8  exception End_of_string;;
9
10 (* ———— flag ———— *)
11 let wc = ref true;; (* true se il termine e' della WC, false se UC *)
12 let constant_pos = ref 3;;
13 let constant_value = ref 3;;
14 let constant_dom = ref 0;;
15 let tree_type = ref 0;; (* 1 per tipo 1, 2 per tipo 2*)
16 let temp_instance_type = ref Type.OneA;; (* tipo delle variabili libere *)
17
18 (* ———— tipi per alberi ———— *)
```

```
19 type lintree =
20   | EmptyL
21   | ZeroL of int * int
22   | OneL of int * int;;
23
24 type rectree =
25   | EmptyR
26   | ZeroR of rectree * rectree
27   | OneR of rectree * rectree;;
28
29 (* ———————— varie ———————— *)
30
31 (* calcola la cardinalita' dei tipi del prim'ordine <>, *, + *)
32 let cardinalita (tipo: Type.t) : int =
33   let rec card (ty: Type.t) =
34     match ty with
35     | Type.SumA (t1, t2) -> (card t1) + (card t2)
36     | Type.TensorA (t1, t2) -> (card t1 ) * (card t2)
37     | Type.Var(a) -> card !temp_instance_type;
38     | Type.ZeroA | Type.OneA -> 1
39     | _ -> failwith "card. not implemented"
40   in card tipo;;
41
42 (*prende in input un tipo del prim'ordine (WC) e restituisce la lista dei
       termini che lo costituiscono *)
43 let rec create_type_list (ty: Type.t)  =
44   let rec makeleft lista  = match lista with
45   | [] -> []
46   | head :: tail -> (Term.mkInlA head) :: makeleft tail
47   and makeright lista = match lista with
48   | [] -> []
49   | head :: tail -> (Term.mkInrA head) :: makeright tail
50   and makepair l1 l2 =
51     (*  prodotto tra un termine e una lista di termini *)
52     let rec makeaux (term: Term.t) (l2: Term.t list) =
53       match l2 with
54       | [] -> []
55       | head :: tail -> Term.mkPairA term head :: makeaux term tail
56     in
57     match l1 with
58     | [] -> []
59     | head:: tail -> (makeaux head l2) @ (makepair tail l2)
60   in
61   match ty with
62   | Type.OneA -> [Term.mkUnitA]
63   | Type.SumA (t1, t2) -> (makeleft (create_type_list t1)) @ (makeright
       (create_type_list t2))
```

```
64    | Type.TensorA (t1, t2) -> makepair (create_type_list t1)
        (create_type_list t2)
65    | Type.Var(a) -> create_type_list !temp_instance_type;
66    | _ -> failwith "create_type list: UC types not implemented";;
67
68 (* cerca TERMINE dentro alle DS e restituisce il suo tipo *)
69 let find_and_print_type (ds: typed_decls) (termine: Term.t): Type.t =
70   let rec findtype d s =
71   match d with
72       | [] -> failwith "Term not found!";
73       | TypedTermDeclA(f, _, a) :: r -> if f = s then a else findtype r s
74         | TypedTermDeclB(f, _, b) :: r -> if f = s then b else findtype r s;
75   in let mytype = (findtype ds (Printing.string_of_termA termine)) in
       mytype;;
76
77 (* trasforma in stringa l'albero ricorsivo in input *)
78 let rec print_tree (treeitem: rectree): string =
79   let buf = Buffer.create 80 in
80     let rec printrec (item: rectree): unit =
81       match item with
82       | EmptyR -> Buffer.add_string buf "empty";
83       | ZeroR (t1, t2) -> (Buffer.add_string buf "0("; printrec t1;
84                           Buffer.add_string buf ","; printrec t2;
                                Buffer.add_string buf ")";)
85       | OneR (t1, t2) -> (Buffer.add_string buf "1("; printrec t1;
86                          Buffer.add_string buf ","; printrec t2;
                               Buffer.add_string buf ")";)
87     in printrec treeitem; Buffer.contents buf;;
88
89 (* trasforma un albero lineare in albero ricorsivo *)
90 let build_tree (lista: (int * lintree) list): rectree =
91  (* prende in input un lintree NODE e la lista dei termini da NODE compreso
       in avanti *)
92  let rec subst (node: lintree) (li: (int * lintree) list): rectree =
93    let rec search_element (n: int) (mylist: (int * lintree) list): lintree =
94      match mylist with
95      | [] -> failwith "search_element: not found";
96      | head:: tail -> if ( (fst head) = n) then (snd head) else
          search_element n tail;
97    in
98    match node with
99    | EmptyL -> EmptyR;
100   | ZeroL (n1, n2) ->
101     (match li with (* faccio in modo che la lista diminuisca ad ogni passo
          *)
102     | [] -> failwith "it can't happen :)";
103     | head :: tail ->  ZeroR( subst (search_element n1 tail) tail, subst
```

```
            (search_element n2 tail) tail);)
104     | OneL (n1, n2) ->
105       (match li with
106       | [] -> failwith "it can't happen :)";
107       | head :: tail -> OneR( subst (search_element n1 tail) tail, subst
            (search_element n2 tail) tail);)
108
109    in
110    match lista with
111    | [] -> EmptyR;
112    | head :: tail -> subst (snd head) lista;;
113
114
115 (* ———————— topsort ——————— *)
116
117 (* prende in input la descrizione dell'albero (lista di coppie int *
      lintree) e restituisce la lista ordinata *)
118 let topsort (lista: (int * lintree) list): (int * lintree) list =
119
120   (* ——— utility per manipolazione liste ——— *)
121   let enqueue elem lista = match lista with
122     | [] -> [elem];
123     | head :: tail -> lista @ [elem];
124   in
125   (* inserisco in coda sse non gia' presente *)
126   let rec enqueue_empty elem lista = match lista with
127     | [] -> [elem];
128     | head :: tail -> if (head = elem) then lista else enqueue_empty elem
          tail;
129   in
130   let rec remove elem lista = match lista with
131     | [] -> []
132     | head :: [] when (head != elem) -> failwith "remove: element not found
          ";
133     | head :: tail -> if (elem = head) then tail
134                       else (if ((List.hd tail) = elem) then head ::
                              (List.tl tail)
135                                 else remove elem tail);
136   in
137   let rec search_node (n: int) (mylist: (int * lintree) list): (int *
        lintree) =
138     match mylist with
139     | [] -> failwith "search element: not found";
140     | head:: tail -> if ( (fst head) = n) then head else search_node n tail;
141   in
142   (* scandisce la lista dei nodi del grafo tree_desc, produce in output la
        lista dell eradici *)
```

```
143  let rec findradix tree_desc elenconodi: (int * lintree) list =
144     let rec israd (n: int) (lista: (int * lintree) list) : bool =
145     match lista with
146     | [] -> true;
147     | head :: tail ->
148  (match (snd head) with
149  | ZeroL (num1, num2) -> if ( (n = num1) || (n = num2)) then false
150        else israd n tail;
151  | OneL (num1, num2) -> if ( (n = num1) || (n = num2)) then false
152        else israd n tail;
153  | EmptyL -> israd n tail;)
154     in
155     match tree_desc with
156     | [] ->  [];
157     | head :: tail -> if (israd (fst head) elenconodi) then (head ::
          findradix tail elenconodi)
158      else findradix tail elenconodi;
159  in
160 (* ciclo principale dell'algoritmo di sorting. produce in output la lista
     ordinata *)
161  let rec toploop (rad_list: (int * lintree) list) sol flush start: (int *
      lintree) list =
162    match (List.length rad_list) with
163    | 0 -> let sol = sol @ flush in sol;
164    | _ -> begin
165     let nodo = (List.hd rad_list) in
166    (* prelevo il primo elemento della lista radici *)
167    ( let rad_list = remove nodo rad_list in
168    (* controllo il val del nodo rimosso *)
169    match (snd nodo) with
170    | EmptyL -> let flush = enqueue_empty nodo flush in
171          toploop rad_list sol flush start;
172    | ZeroL (num1, num2) ->
173      (* inserisco i figli in rad_list e il nodo nella sol*)
174      let rad_list = enqueue (search_node num1 start) rad_list  in
175      let rad_list = enqueue (search_node num2 start) rad_list in
176      let sol = enqueue nodo sol in
177      toploop rad_list sol flush start;
178    | OneL (num1, num2) ->
179      (* inserisco i figli in rad_list e il nodo nella sol*)
180      let rad_list = enqueue (search_node num1 start) rad_list in
181      let rad_list = enqueue (search_node num2 start) rad_list in
182      let sol =  enqueue nodo sol in
183      toploop rad_list sol flush start;)
184    end;
185  in
186
```

```
187   let radix_queue = findradix lista lista in
188   if ((List.length radix_queue) > 1) then failwith "Error: found more than
        1 radix"
189   else if ((List.length radix_queue) = 0) then failwith "Error, direct
        cycle detected"
190   else
191   let newsol = toploop radix_queue [] [] lista in newsol;;
192
193 (* ---------- valutazione delle stringhe ---------- *)
194
195 (* valuta una stringa della WC. Parametri: la funzione da valutare e le
      liste di tutti i suoi possibili input e output *)
196 let eval_string_WC (query: Term.t) (lista_input: Term.t list)
      (lista_output: Term.t list): string =
197   let buf = Buffer.create 80
198   in
199   let map_out (output: Term.t) (lista: Term.t list): int =
200     (let rec search_term termine ll pos =
201     match ll with
202     | head :: tail -> if (termine = head) then pos
203                       else (search_term termine tail (pos+1));
204     | [] -> failwith "error: map_out"
205     in let posizione = search_term output lista 0 in posizione)
206   in
207   let valuta (termine: Term.t): unit =
208     let valutazione = (Evaluation.eval_closed termine) in
209       match valutazione  with
210       | Some v -> let pos = map_out v lista_output in
211         if (pos = 2) then (* se pos = 2 la stringa e' finita *)
212           (Buffer.add_string buf "(e)"; raise End_of_string)
213         else (Buffer.add_string buf (string_of_int pos);
214               Buffer.add_string buf " ");
215       | None -> failwith "Unexpected None in evaluation";
216   in
217   let rec evalW (qu: Term.t) (li: Term.t list) (lo: Term.t list): unit =
218     match li with
219     | [] -> (); (* finita valutazione *)
220     | head :: tail -> (valuta (Term.mkAppA qu head));  evalW qu tail lo;
221
222   in (try evalW query lista_input lista_output with End_of_string -> ());
223   Buffer.contents buf;;
224
225 (* valuta una stringa della UC. Parametri: la funzione da valutare e le
      liste di tutti i suoi possibili input e output *)
226 let eval_string_UC (query: Term.t) (lista_input: Term.t list)
      (lista_output: Term.t list): string =
227   let buf = Buffer.create 80
```

```
228  in
229  let map_UC_out (output: Term.t) (lista: Term.t list): int =
230    (let rec search_term termine ll pos =
231      match ll with
232      | head :: tail ->
233         (match head.desc with
234         | Term.InlA(t) -> search_term termine tail pos;
235         | Term.InrA (t) -> if (termine = head) then pos
236                            else (search_term termine tail (pos+1));
237         | _ -> failwith "map_UC_out: error";)
238      | [] -> failwith "output not found in map_UC_out";
239    in
240    match output.desc with
241    | Term.InlA(t) -> 3; (* la stringa non e' costante.*)
242    | _ -> let posizione = search_term output lista 0 in posizione;)
243  in
244  (* interroga query con inr(<>), salva l'output 0/1/2/3 nella var globale
       constant_value  *)
245  let save_first_value (query: Term.t) (lista_output: Term.t list): unit =
246    (let first_output_term =
247      Evaluation.eval_closed (Term.mkAppA query
           (Term.mkInrA(Term.mkUnitA))) in
248       match first_output_term with
249       | Some v -> let pos = (map_UC_out v lista_output) in
              (constant_value := pos);
250       | None -> failwith "error: is_constant_string\n";)
251  in
252  let valuta (termine: Term.t): unit =
253    let valutazione = (Evaluation.eval_closed termine) in
254      match valutazione  with
255      | Some v  -> let output = (map_UC_out v lista_output) in
256                   if (output = 2) then (Buffer.add_string buf "(e)";
257                                         raise End_of_string)
258                   else (Buffer.add_string buf (string_of_int output);
259                         Buffer.add_string buf " ");
260      | None -> failwith "Error: valuta(evalUC)";
261  in
262  let rec build_constant_string (acc: int): unit =
263    (match acc with
264    | 0 -> Buffer.add_string buf "(constant)";
265    | _ -> Buffer.add_string buf (string_of_int !constant_value);
266           Buffer.add_string buf " "; build_constant_string (acc-1);)
267  in
268  let rec evalRecUC (qu: Term.t) (li: Term.t list) (lo: Term.t list): unit =
269    match li with
270    | [] -> (); (* non ci arriviamo mai *)
271    | head :: tail when (head = (Term.mkInrA(Term.mkUnitA))) -> ();
```

```
272                     (* inr(<>) era il primo val che abbiamo dato in input *)
273     | head :: tail -> if (!constant_value != 3)  (* stringa costante *)
274                       then  build_constant_string !constant_dom
275                       else                        (* stringa non costante  *)
276                       (valuta (Term.mkAppA qu head); evalRecUC qu tail lo);
277   in
278   let () = save_first_value query lista_output in
279     (try evalRecUC query lista_input lista_output with End_of_string -> ());
280   Buffer.contents buf;;
281
282 (* ———————— valutazione degli alberi ———————— *)
283
284 (* valuta il termine della WC query *)
285 let eval_tree_WC (query: Term.t) (in_list: Term.t list): string =
286   (* mappa il termine NUMERO nell'intero corrispondente, cercandolo nella
        lista degli input *)
287   let mappa (numero: Term.t): int =
288     let rec map_number (numb: Term.t) (acc: int) (li: Term.t list): int =
289       match li with
290       | [] -> failwith "map_number: input not found";
291       | head :: tail -> if (head.desc = numb.desc) then acc
292                         else (map_number numb  (acc+1) tail);
293     in map_number numero 0 in_list
294   in
295   (* trasforma la coppia in input in coppia int * lintree *)
296   let map_pair (coppia: Term.t * Term.t): (int * lintree) =
297     let first = mappa (fst coppia) in
298       let second =
299         if (!tree_type = 1) then
300           (match (snd coppia).desc with
301           | Term.InrA (_) -> EmptyL;
302           | Term.InlA (t1) ->
303               (match t1.desc with
304               | Term.InlA(pair) ->
305                   (match pair.desc with
306                   | Term.PairA (c1,c2) -> ZeroL(mappa c1, mappa c2);
307                   | _ -> failwith "pair expected in map_pair";)
308               | Term.InrA(pair) ->
309                   (match pair.desc with
310                   | Term.PairA (c1,c2) ->  OneL(mappa c1, mappa c2);
311                   | _ -> failwith "pair expected in map_pair";)
312               | _ -> failwith "type mismatch 1 in map_pair";)
313           | _ -> failwith "output type mismatch in map_pair";)
314         else if (!tree_type = 2) then
315           (match (snd coppia).desc with
316           | Term.InlA (pair) ->
317               (match pair.desc with
```

```
318              | Term.PairA (c1, c2) -> ZeroL (mappa c1, mappa c2);
319              | _ -> failwith "pair expected in map_pair";)
320          | Term.InrA (t) ->
321              (match t.desc with
322              (* puo' essere inl(n*n) oppure inr(<>) *)
323              | Term.InrA(_) -> EmptyL;
324              | Term.InlA (pair) ->
325                 (match pair.desc with
326                 | Term.PairA (c1,c2) -> OneL(mappa c1, mappa c2);
327                 | _ -> failwith "pair expected in map_pair";)
328              | _ -> failwith "output type mismatch 2 in map_pair";)
329          | _ -> failwith "output type mismatch 3 in map_pair";)
330        else failwith "unknow tree type in map_pair";
331    in (first, second)
332  in
333  (* restituisce la coppia (input * output) mappata in (int * lintree)  *)
334  let valuta (funct: Term.t) (input: Term.t) =
335    let valutazione = Evaluation.eval_closed (Term.mkAppA funct input) in
336      match valutazione with
337      | Some v ->  map_pair (input, v);
338      | None -> failwith "valuta: None inside evaluation";
339  in
340  (* produce la lista completa di coppie  input-output*)
341  let rec evalWT (qu: Term.t) (li: Term.t list): (int * lintree) list  =
342  match li with
343  | [] -> [];
344  | head :: tail -> (valuta qu head) ::  evalWT qu tail;
345  in
346  let output_list = evalWT query in_list in
347     let sorted_output = topsort output_list in
348       let result = build_tree sorted_output in (print_tree result);;
349
350 (* valuta l'albero della UC query *)
351 let eval_tree_UC (query: Term.t) (in_list: Term.t list): string  =
352  (* trasforma la coppia (Term.t Term.t) in  (int * lintree) *)
353  let map_output_UC (coppia: Term.t * Term.t) (lis: Term.t list): int *
      lintree  =
354    let map_n_UC (numb: Term.t) (l: Term.t list): int =
355      let rec mappainput (n: Term.t) (ly: Term.t list) (acc: int): int =
356        match ly with
357        | [] -> failwith "map_n_UC: input not found";
358        | head :: tail ->
359            (match head.desc with
360            (* o sono inl di coppie oppure e' l'ultimo termine, inr (<>) *)
361            | Term.InrA(_) -> failwith "map_n_UC: found inr(<>)";
362            | Term.InlA (pair) ->
363                (match pair.desc with
```

```
364                 | Term.PairA(t1,t2) -> if t2.desc = n.desc then acc else
                        mappainput n tail (acc+1);
365                 | _ -> failwith "mappainput";)
366            | _ -> failwith "map_n_UC: term";)
367       in mappainput numb l 0;
368 in
369 (* mappa in numeri termini del dominio *)
370 let map_inputUC (inp: Term.t) (ly: Term.t list): int =
371   let rec mi (i: Term.t) (li: Term.t list) (acc: int): int =
372     match li with
373     | [] -> failwith "map_inputUC: input not found";
374     | head :: tail ->  if (head.desc = i.desc) then acc else mi i tail
          (acc + 1);
375   in mi inp ly 0;
376 in
377 let first = map_inputUC (fst coppia) lis in
378   let second =
379     begin
380     if (!tree_type = 1) then
381        (match (snd coppia).desc with
382        | Term.InrA(t1) ->
383            (match t1.desc with
384            | Term.InrA(_) -> EmptyL;
385            | Term.InlA(t2) ->
386                (match t2.desc with
387                (* pair e' PairA (t3,t4), t3 e t4 sono i termini che
                       dobbiamo cercare nell'input *)
388                | Term.InlA (pair) ->
389                    (match pair.desc with
390                    | Term.PairA(n1,n2) -> ZeroL((map_n_UC n1 lis),
                           (map_n_UC n2 lis));
391                    | _ -> failwith "mapUc 6";)
392                | Term.InrA(pair) ->
393                    (match pair.desc with
394                    | Term.PairA(n1,n2) -> OneL((map_n_UC n1 lis),
                           (map_n_UC n2 lis));
395                    | _ -> failwith "mapUc t";)
396                | _ -> failwith "mapUC 2";)
397            | _ -> failwith "mapUC 3";)
398      (* | Term.InlA(pair) -> impossibile, la prima richiesta viene eseguita
           a parte *)
399        | _ -> failwith "pamUC 4";)
400     else if (!tree_type = 2) then
401        (match (snd coppia).desc with
402        | Term.InrA(t1) ->
403            (match t1.desc with
404            | Term.InlA(pair) ->
```

```
405                 (match pair.desc with
406                 | Term.PairA (n1,n2) -> ZeroL((map_n_UC n1 lis), (map_n_UC
                        n2 lis));
407                 | _ -> failwith "mapUC 5";)
408             | Term.InrA (term) ->
409                 (match term.desc with
410                 | Term.InlA(pair) ->
411                     (match pair.desc with
412                     | Term.PairA (n1,n2) -> OneL((map_n_UC n1 lis),
                            (map_n_UC n2 lis));
413                     | _ -> failwith "mapUC 6";)
414                 | Term.InrA(_) -> EmptyL;
415                 | _ -> failwith "mapUC 5";)
416       | _ -> failwith "mapUC 7";)
417       (* | Term.InlA(pair) -> Printf.printf " inl(<1,1>)\n" : impossibile *)
418   | _ -> failwith "mapUC 8";)
419     else failwith "unknown UC tree type"
420     end
421   in (first, second);
422 in
423 let valutaU (funct: Term.t) (input: Term.t): int * lintree =
424   let valutazione = Evaluation.eval_closed (Term.mkAppA funct input) in
425     match valutazione with
426     | Some v -> map_output_UC (input, v) in_list;
427     | None -> failwith "valuta: None inside evaluation";
428 in
429 (* produce una lista di coppie input-output di tipo int * lintree.
430 Si ferma quando arriva all'ultimo termine, inr(<>), che viene valutato per
        primo
431 tramite la funzione first_query*)
432 let rec evalUT (qu: Term.t) (li: Term.t list): (int * lintree) list =
433   match li with
434   | [] -> [];
435   | head :: tail when (head = (Term.mkInrA(Term.mkUnitA))) -> [];
436   | head :: tail -> (valutaU qu head) ::  evalUT qu tail;
437 in
438 (* rest. true se la funzione non e' costante *)
439 let first_query (tree: Term.t): bool =
440   let reply = Evaluation.eval_closed (Term.mkAppA tree
        (Term.mkInrA(Term.mkUnitA))) in
441     (match reply with
442     | Some v ->
443         (match v.desc with
444         (* se risponde inl significa continua, se inr e' costante *)
445         | Term.InlA(_) ->  true;
446         | Term.InrA(_) -> false;
447         | _ -> failwith "first_query: reply";)
```

```
448       | None -> failwith "first_query: none";)
449  in
450    if (first_query query) then
451      (let output_list = evalUT query in_list in
452        let sorted_output = topsort output_list in
453          let result = build_tree sorted_output in (print_tree result);)
454    else failwith "tree gave a constant response: not implemented";;
455
456 (* ———————— funzioni di riconoscimento stringhe/alberi ———————— *)
457
458 (* restituisce TRUE se il tipo corrisponde a una stringa *)
459 let is_a_string (tipo: Type.t): bool =
460   (* stringa WC ha tipo FunA (dom,cod), calcoliamo la cardinalita' di cod *)
461   (* stringa UC e' FunA(dom, cod) con cod = SumA(TensorA(_,OneA), truecod)),
462   calcoliamo la cardinalita' di truecod *)
463   match tipo with
464   | Type.FunA (dom, cod) ->
465     if (!wc) then          (* WC query *)
466       (let codominio = cardinalita cod in
467         if ((codominio = 2) || (codominio = 3)) then true else false;)
468     else                   (* UC query *)
469     (match cod with
470     | Type.SumA (fst, snd) ->
471       (* fst = (1*'b) se e' una stringa normale, = 'a0 se e' una stringa
          costante *)
472         let codominio = cardinalita snd in
473           if ( (codominio = 2) || (codominio = 3) ) then true else false;
474     | _ ->  false; (* il cod non e' un Sum *) )
475   | _ -> failwith "error: is_a_string";;
476
477
478 (* restituisce TRUE se il tipo rappresenta un albero, salvando nella var.
     globale tree_type il suo tipo (1 oppure 2) *)
479 let is_a_tree (ty: Type.t) : bool =
480  (* verifica uguaglianza fra tipi del prim'ordine: rest. true se i tipi
      sono uguali *)
481  let rec compare (first: Type.t) (snd: Type.t): bool =
482     match first with
483     | Type.Var(_) -> (match snd with
484                       | Type.Var(_) -> true; | _ -> false;)
485     | Type.OneA -> if (snd = Type.OneA) then true else false;
486     | Type.ZeroA -> if (snd = Type.ZeroA) then true else false;
487     | Type.TensorA (t1,t2) -> (match snd with
488                               | Type.TensorA (t3,t4) -> ((compare t1 t3) &&
                                    (compare t2 t4));
489                               | _ -> false;)
490     | Type.SumA (t1,t2) -> (match snd with
```

```
491                              | Type.SumA (t3, t4) ->  ((compare t1 t3) &&
                                    (compare t2 t4));
492                              | _ -> false;)
493      | _ -> failwith "higher−order compare not implemented";
494   in
495   match ty with
496   | Type.FunA (dom, cod) ->
497        if (!wc) then      (* WC *)
498         (let prod = Type.TensorA (dom, dom)  (* prod = n*n *) in
499            (* tipo1 : SumA(SumA(prod, prod), OneA) *)
500            let proof1 = Type.SumA(Type.SumA(prod,prod), Type.OneA) in
501              if (compare cod proof1) then (tree_type :=1; true)
502              else (* tipo2: SumA(prod, SumA(prod, OneA)) *)
503               (let proof2 = Type.SumA(prod, Type.SumA(prod, Type.OneA)) in
504                 if (compare cod proof2) then (tree_type :=2; true) else
                       false;))
505        else              (* UC *)
506        (match dom with (* dom UC = (1 * domWC) + 1 *)
507        | Type.SumA (fst, _) ->
508            (match fst with
509            | Type.TensorA(_, realdom) ->
510              let prod = Type.TensorA (realdom, realdom) in  (* n*n *)
511                (match cod with
512                (* tipo 1: SumA (1*1, SumA(TensorA(1,1),
                       SumA(SumA(prod,prod), OneA))) *)
513                | Type.SumA (t1, t2) ->
514                  let proof1 = Type.SumA(Type.SumA(prod,prod), Type.OneA) in
515                    (if (compare proof1 t2) then (tree_type :=1; true)
516                     else
517                     (* tipo 2: SumA(prod, SumA(prod, Type.OneA) *)
518                       (let proof2 = Type.SumA(prod, Type.SumA(prod,
                             Type.OneA)) in
519                         (if (compare proof2 t2) then (tree_type :=2; true)
                               else false; )))
520                | _ -> false;)
521            | _ -> failwith "is_a_tree";)
522        | _ -> failwith "";)
523    | _ -> failwith "";;
524
525
526 (* ────────── main ────────── *)
527
528 (* parametri DS (dichiarazioni del file tipate) QUERY (la query) INST_VALUE
        (il valore di instance_type) *)
529 let explore_main (ds: typed_decls) (query: Term.t) (inst_value: Type.t):
        unit =
530   temp_instance_type := inst_value;  (* tipo da associare alle var libere *)
```

```
531   constant_dom := 0;                    (* per le stringhe costanti *)
532   tree_type := 0;                       (* tipo degli alberi *)
533
534   (* controllo che la query sia contenuta nelle definzioni
535   e verifico se si tratta di un termine della UC o della WC *)
536   let query_type = find_and_print_type ds query in
537   (match query_type with
538   | Type.FunA(_,_) -> wc := true;
539   | Type.FunB (_,_,_) -> wc := false;
540   | _ -> failwith "Error: #explore parameter must be a function");
541
542   (* compilo la query *)
543   let ground_query, closed_query_type = produce_compiled_query query ds in
544
545   (* controllo che sia una stringa *)
546   if (is_a_string closed_query_type) then
547     (* creazione dei termini e alla valutazione *)
548     (match closed_query_type with
549     | Type.FunA (dom, cod) ->
550         let listOfVal = create_type_list dom in (* lista degli input *)
551         (* in caso di stringhe costanti salvo la cardinalita' del dominio *)
552         if (!wc = false) then
553           match dom with
554           | Type.SumA(t1,t2) -> constant_dom := cardinalita t1;
555           | _ -> ();
556         else (); (* NOP *)
557         let listOfOutput = create_type_list cod in
558           (if (!wc) then
559            Printf.printf "\n: string\n= %s\n" (eval_string_WC ground_query
                   listOfVal listOfOutput)
560            else
561            Printf.printf "\n: string\n= %s\n" (eval_string_UC ground_query
                   listOfVal listOfOutput));
562     | _ -> failwith "error: main")
563
564  (* altrimenti controllo che sia un albero *)
565   else
566   ( if (is_a_tree closed_query_type) then
567       (match closed_query_type with
568         | Type.FunA(t1,t2) ->
569            (let listOfInput = create_type_list t1 in
570             if (!wc) then Printf.printf": tree\n= %s\n" (eval_tree_WC
                   ground_query listOfInput)
571             else Printf.printf"\n: tree\n= %s\n" (eval_tree_UC ground_query
                   listOfInput))
572         | _ -> failwith "tree main";)
573
```

```
574   (* non e' stringa ne' albero *)
575   else failwith "unable to detect a string or a tree");;
```

5.5 Note sulla Realizzazione del Codice di #explore

In questo capitolo abbiamo illustrato in dettaglio i tipi di dato che volevamo sottoporre ad esplorazione e le procedure che abbiamo messo in atto per implementare `#explore`. Per realizzare questa nuova funzionalità abbiamo creato un modulo separato dal resto del codice dell'interprete, in modo da facilitare non solo l'attuale fase di testing degli algoritmi ma anche evetuali interventi di manutenzione che potrebbero essere necessari in futuro.
Come si può vedere scorrendo il codice del `main`, abbiamo articolato il codice in tre grandi unità logiche, agglomerando le funzioni appartenenti a ciascuna unità:

1. riconoscimento della funzione;
2. creazione della lista degli input;
3. valutazione della funzione.

In particolare, abbiamo progettato le funzioni di valutazione - le più lunghe e laboriose - in modo che abbiano un unico punto di ingresso e di uscita: un programmatore che in futuro dovesse apportare delle modifiche avrà così il vantaggio di potre lavorare su sezioni circoscritte di codice e individuare in maniera immediata la relazione e il passaggio dei parametri fra le diverse funzioni.
Dal punto di vista della tecnica di programmazione, abbiamo cercato di sfruttare il più possibile la natura funzionale di OCaml, limitando al massimo l'impego di *features* di tipo imperativo; gli unici costrutti imperativi utilizzati sono le variabili `ref` che abbiamo usato come flag per orientare i pattern

matching (ad esempio, la variabile `wc` che registra la classe di appartenenza della *query*, semplificando il pattern matching eseguito per i controlli sul tipo della funzione) e l'eccezione `End_of_string` che solleviamo per terminare in modo elegante la computazione quando si incontra lo speciale valore che indica la fine della stringa.
La struttura del codice è molto ripetitiva: per manipolare i dati e comunicarli tra le funzioni abbiamo ovunque usato le liste, e funzioni che operano su di esse ricorsivamente; per gli onerosi controlli sui tipi abbiamo fatto ovunque ricorso al pattern matching, che riteniamo un costrutto particolarmente leggibile e chiaro.

Bibliografia

DLS10a U. Dal Lago and U. Schöpp, Functional programming in sublinear space. *European Symposium on Programming (ESOP 2010), Cyprus, LNCS 6012, ©Springer-Verlag*, 2010

DLS10b U. dal Lago and U. Schöpp, Type inference for sublinear space functional programming. *ASIAN Symposium on Programming Languages and Systems (APLAS 2010), Shanghai, China*, 2010

F10 F. Foschini, Programmazione funzionale in spazio logaritmico: funzioni di ordinamento, *Tesi di Laurea in Sicurezza e Crittografia*, Bologna aa. 2009/2010.

GM06 M. Gabbrielli e S. Martini, Linguaggi di programmazione. McGraw-Hill, 2006

L10 M. Lodi, Programmazione funzionale in spazio logaritmico: una libreria di funzioni aritmetiche, *Tesi di Laurea in Sicurezza e Crittografia*, Bologna aa. 2009/2010.