Alma Mater Studiorum - Università di Bologna

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

PROGRAMMAZIONE FUNZIONALE IN SPAZIO LOGARITMICO: UNA LIBRERIA DI FUNZIONI ARITMETICHE

Tesi di Laurea in Sicurezza e Crittografia

Relatore:
Ill.mo Dott. Ugo Dal Lago

Presentata da:
Michael Lodi

II Sessione
Anno Accademico 2009/2010

A chi mi vuole bene davvero

Introduzione

"I'm not small, I'm space-efficient."
– Rachael Leigh

Negli ultimi decenni la capacità dei dispositivi di memorizzazione dei calcolatori (memoria primaria e secondaria) è aumentata notevolmente, di pari passo con la diminuzione dei prezzi di tali dispositivi. Inoltre l'introduzione di tecniche quali la cosiddetta *memoria virtuale* o la *cache* ha permesso di avere uno spazio sempre maggiore per memorizzare i programmi, i dati e i risultati temporanei della computazione associata.

È naturale quindi che l'attenzione si sia concentrata soprattutto sull'ottimizzazione del *tempo* di esecuzione, sia a livello di algoritmi che di linguaggi di programmazione.

Vari sono i campi in cui però è necessario garantire l'utilizzo di una quantità fissata (magari sensibilmente più piccola dell'input) di memoria:

- le computazioni in cui la garanzia di proprietà formali è *critica*, per esempio i programmi per la gestione delle transazioni bancarie o i programmi che regolano il funzionamento degli aerei o delle centrali elettriche;

- i dispositivi portatili o integrati (dai cellulari fino ai microcontrollori per gli elettrodomestici) in cui la memoria è limitata e non c'è possibilità di usare memorie ausiliarie come cache o dischi, indispensabili per l'implementazione di tecniche quali la *memoria virtuale*;

- nel *data-mining* (che si occupa sostanzialmente dell'estrazione di una *informazione* a partire da grandi quantità di dati attraverso metodi automatici o semi-automatici, effettuando su tali dati operazioni di aggregazione *statistica*), in cui ovviamente non è possibile far risiedere tutti i dati in memoria centrale;

- nel *web-crawling*, che si occupa specificamente di scandagliare periodicamente il web, di solito per conto di un motore di ricerca, e indicizzare i contenuti delle pagine reperendo più informazioni possibili su di esse, per poi restituirli quando vengono ricercati: in questo caso l'input è addirittura l'intero *World Wide Web*;

- più in generale, in tutte quelle computazioni con dati che risiedono su computer diversi da quello che sta eseguendo il programma;

- in tutte le applicazioni che non lavorano con un input prefissato, ma che ricevono - solitamente a ritmi molto elevati - stream potenzialmente infiniti di dati: in queste situazioni l'utilizzo di spazio non può dipendere in alcun modo dalla lunghezza dell'input (se così fosse infatti si occuperebbe presto l'intera memoria disponibile anche solo incrementando un contatore per ogni input ricevuto).

In questa tesi non ci proponiamo di fornire una risposta concreta a nessuno di questi problemi, cercheremo invece di studiare (e, in piccola parte, anche di estendere) alcuni strumenti introdotti recentemente, che possono gettare le basi per lo sviluppo di applicazioni indispensabili nei campi citati.

Inizieremo dunque con lo studiare (Cap. 1) le nozioni di Informatica Teorica, e in particolare di Teoria della Complessità Computazionale, che stanno dietro ai limiti spaziali di un problema. Introdurremo dunque le nozioni di base: il modello della Macchina di Turing e la definizione di *classi di complessità*. Presenteremo poi un modello alternativo a quello della Macchina di Turing, che di solito si usa per le dimostrazioni che riguardano i limiti in spazio, chiamato *Macchina di Turing Offline*. Questo modello, pur avendo

lo stesso potere espressivo della Macchina di Turing, ci permette di analizzare algoritmi che utilizzano spazio sub-lineare rispetto ai dati in ingresso: l'input e l'output non sono memorizzati, ma richiesti e restituiti da e verso l'ambiente in maniera *interattiva*.

Questo strumento ci permetterà di passare a un nuovo "modo" di pensare alla computazione: la *programmazione bidirezionale*. Sarà però subito chiaro che questo stile richiede di modificare, in maniera abbastanza radicale, il modo di pensare agli algoritmi e ai programmi, dovendo prestare attenzione a nuovi e diversi aspetti.

Un linguaggio di programmazione di recentissima introduzione, IntML, si propone di superare questi limiti fornendo al programmatore delle primitive per programmare in modo "naturale", demandando poi al linguaggio stesso la *traduzione* da stile unidirezionale a stile bidirezionale. Viene inoltre dimostrato che, grazie a questo modello, IntML permette di scrivere programmi che utilizzano uno spazio logaritmico di memoria rispetto all'input.

IntML è definito formalmente e in modo teorico nei lavori che lo introducono [DLS10a, DLS10b], ma possiede anche un'implementazione concreta. Esiste infatti un interprete, ancora prototipale, che permette di utilizzare il linguaggio per realizzare programmi *logspace*.

Sarà compito di questa tesi dunque realizzare un "manuale" (Cap. 2) che si focalizzi sopratutto sugli aspetti di sintassi e semantica concreta di IntML, tralasciando magari i dettagli implementativi e formali che, pur essendo fondamentali, rischiano di confondere il programmatore abituato ad usare linguaggi tradizionali. Analizzeremo quindi nel dettaglio i vari costrutti che l'implementazione fornisce e mostreremo semplici esempi su come utilizzarli concretamente.

Verrà naturale voler estendere il linguaggio con una serie di features utili, nella forma di una *libreria di funzioni,* che metta a disposizione del programmatore alcune routine basilari, ma non presenti nel linguaggio, di modo che egli possa poi utilizzarle direttamente per scrivere altri programmi, avendo la garanzia del loro uso limitato di spazio. In particolare, concentreremo la

nostra attenzione (Cap. 3) su alcune *funzioni aritmetiche* che possono essere calcolate in *spazio logaritmico.*

Analizzeremo infine quali aspetti devono ancora essere sviluppati, dal punto di vista dello studio teorico della complessità, dell'usabilità del linguaggio e degli strumenti a supporto della programmazione.

Indice

Capitolo 1

Cosa c'è e Cosa ci Serve

> *The idea behind digital computers may be explained by saying that these machines are intended to carry out any operations which could be done by a human computer. The human computer is supposed to be following fixed rules; he has no authority to deviate from them in any detail. We may suppose that these rules are supplied in a book, which is altered whenever he is put on to a new job. He has also an unlimited supply of paper on which he does his calculations.*
>
> – Alan Turing

In questo capitolo cercheremo inizialmente di capire perché sia necessario, o conveniente, utilizzare un linguaggio che garantisca l'uso di uno spazio di memoria sensibilmente inferiore a quello che servirebbe per memorizzare l'input.

Vedremo poi una carrellata sulle nozioni teoriche e tecniche importanti per comprendere il seguito della trattazione: qualche accenno sulla programmazione funzionale, una breve introduzione alla complessità computazionale tramite la nozione di Macchina di Turing, e infine lo studio di una particolare classe di complessità su cui concentreremo l'attenzione in tutto il lavoro. Introdurremo infine il concetto di programmazione bidirezionale.

1.1 Programmazione in Spazio Limitato

Come già detto, molteplici sono i campi in cui è necessario garantire l'utilizzo di una quantità fissata (magari sensibilmente più piccola dell'input) di memoria.

Per meglio comprendere il problema, analizziamo il seguente esempio: è sempre più frequente la situazione in cui un programmatore si trovi a dover interagire, nella realizzazione di un software, con una collezione di dati molto grande, magari memorizzata su un server remoto. Utilizzando un approccio tradizionale, per scrivere una funzione che lavora su alcuni dei dati contenuti sul server, senz'altro *l'intera collezione* dovrebbe essere input della funzione (e quindi, ovviamente, trasferita interamente sulla macchina locale). Come si può immaginare, questo porterebbe ben presto a problemi (es. *stack overflow*) dovuti alla mancanza di spazio sull'elaboratore che sta eseguendo la computazione.

Un classico approccio al problema consiste nel passare alla funzione non l'intera collezione, bensì *un puntatore ad essa* (si legga, in questo caso, *un URL*). A questo punto sarà compito della funzione *interrogare* la collezione di dati per reperire solo le informazioni di cui ha bisogno.

Questo approccio, sebbene risolutivo, modifica radicalmente lo stile che il programmatore deve adottare: si passa così da una programmazione unidirezionale a una *bidirezionale*.

Come di solito si fa in Informatica, il passo successivo è quello di aggiungere un livello di astrazione che nasconda la complessità della programmazione bidirezionale, permettendo di scrivere software nel modo tradizionale.

Una proposta che va in questa direzione verrà presentata nel Capitolo 2.

1.2 Programmazione Funzionale

Il linguaggio che presenteremo utilizza quello che viene chiamato *paradigma funzionale*.

Come ben descritto in [MG06], la principale caratteristica di un linguaggio funzionale puro è quello di non possedere il concetto di memoria, e quindi di effetto collaterale. I linguaggi di programmazione convenzionali basano la loro computazione sulla trasformazione dello *stato*: centrali sono quindi i concetti di *variabile modificabile* e il costrutto di *assegnamento*.

Il paradigma funzionale invece (che ha il suo precursore in un modello teorico chiamato *lambda-calcolo*) basa la computazione sulla *riscrittura* dei valori: le modifiche hanno luogo solo nell'ambiente e non c'è necessità della nozione di memoria. Di conseguenza non ha senso parlare di iterazione, e centrale diviene invece il concetto di ricorsione.

Sintatticamente parlando, il linguaggio non ha comandi, ma solo *espressioni*. I due costrutti principali sono l'astrazione di una variabile rispetto a un'espressione e l'applicazione di una espressione (che denota una funzione) a un'altra espressione che ne denota l'argomento.

Non ci sono vincoli sulla possibilità di passare funzioni come argomento ad altre funzioni o di restituirle come risultato: c'è dunque una *perfetta omogeneità tra programmi e dati.*

Nel seguito supporremo che il lettore conosca i principi che regolano il funzionamento dei linguaggi di programmazione, e che abbia padronanza con linguaggi funzionali quali Scheme o ML[1].

1.3 Complessità Computazionale

La teoria della **complessità computazionale** studia le risorse minime necessarie (principalmente tempo di calcolo e memoria) per la risoluzione di un problema. I problemi vengono così classificati in differenti *classi di complessità*, a seconda di quanto il migliore algoritmo di risoluzione noto sia efficiente.

[1]Per un'introduzione completa a Scheme si veda [FFFK03], mentre per un'introduzione a (un dialetto di) ML si veda [CM98].

Per misurare l'efficienza di un algoritmo in maniera univoca bisogna definire una metrica indipendente dalle tecnologie utilizzate. Per questo motivo si utilizza un *modello di calcolo astratto*, ovvero la Macchina di Turing (abbreviato MdT).

1.3.1 Macchina di Turing

Intuitivamente, possiamo descrivere una MdT come costituita da un nastro infinito, diviso in celle. I simboli sul nastro appartengono a un fissato alfabeto Σ. Ogni cella è inizializzata, al limite con un carattere convenzionale "blank". Esiste poi una testina di lettura che legge una cella alla volta. La macchina è inoltre controllata da un *automa a stati finiti*.

Il funzionamento intuitivo è il seguente: la macchina si trova in uno stato, legge un carattere e, a seconda dello stato e del carattere letto, compie una determinata azione. Formalmente, si può dare la seguente:

Definizione 1.1 (Macchina di Turing)**.** Si definisce Macchina di Turing una quadrupla del tipo

$$T = (Q, \Sigma, \delta, q_0)$$

dove:

- Q è un insieme finito di stati;
- Σ è l'alfabeto del nastro;
- δ è una *funzione di transizione* (moralmente: il **programma** della MdT);
- $q_0 \in Q$ è lo stato iniziale;
- l'input del programma è sul nastro all'inizio della computazione;
- l'output del programma si trova sul nastro alla fine della computazione.

La funzione δ ha tipo $Q \times \Sigma \longrightarrow Q \times \Sigma \times \{L, R\}$.

Così, se la macchina si trova in un certo stato q e sta leggendo il simbolo a, allora se $\delta(q, a) = (r, b, m)$ la macchina sovrascriverà il simbolo a con il simbolo b e andrà nello stato r. A questo punto muoverà la testina a sinistra o a destra, a seconda che il valore di m sia rispettivamente L o R.

Nella nostra definizione, per ogni q e a, δ assume un solo valore (è dunque una *funzione*): in questo caso si parla di *Macchina di Turing deterministica.*

Se invece, per ogni coppia stato-carattere, δ ammette zero o più di un valore (è quindi una *relazione*) si parla di *Macchina di Turing non deterministica.*

Durante una computazione, avvengono dei cambiamenti nello stato corrente dell'automa, nel contenuto del nastro e nella posizione della testina. Una *tripla* che contiene queste tre informazioni è detta **configurazione istantanea** della MdT.

Esistono due particolari configurazioni, dette **configurazione di accettazione** e **configurazione di rigetto**: se la macchina raggiunge una di queste, la computazione si arresta (e si dice che la macchina *accetta* o *rifiuta* la stringa in input). Se non le raggiunge mai, si dice che la macchina entra in *loop*.

MdT come Funzione

Una MdT prende in input una stringa e restituisce una stringa, dunque può essere vista come una funzione da Σ^* a Σ^* [2].

Possiamo quindi dare la seguente

Definizione 1.2 (Linguaggio riconosciuto)**.** L'insieme delle *stringhe* che una MdT accetta è il **linguaggio riconosciuto** da quella macchina.

[2]Con Σ^* intendiamo, utilizzando l'operatore *Stella di Kleene,* le successioni di lunghezza finita di elementi di Σ, comprendendo anche la successione vuota.

MdT Multinastro

Una **Macchina di Turing Multinastro** è del tutto simile a una MdT tradizionale, ma ha in dotazione diversi (k) nastri. La definizione rimane invariata, a parte per la funzione di transizione che sarà

$$\delta : Q \times \Sigma^k \longrightarrow Q \times \Sigma^k \times \{L,\, R,\, S\}^k$$

Ad esempio

$$\delta(q_i,\, a_1, ...,\, a_k) = (q_j,\, b_1, ...,\, b_k,\, m_1, ...,\, m_k)$$

significa che, se la macchina si trova nello stato q_i e le testine da 1 a k leggono i simboli $a_1...a_k$, allora la macchina va nello stato q_j, scrive i simboli $b_1...b_k$ e muove rispettivamente le testine a destra, sinistra, o le mantiene ferme, a seconda che $m_1...m_k$ valgano rispettivamente R, L o S.

A prima vista può sembrare che la MdT Multinastro sia più espressiva di quella ordinaria, ma si può dimostrare che entrambe hanno in realtà *lo stesso potere espressivo.*

1.3.2 Perché il Modello Non Conta: la *Tesi di Church-Turing*

La MdT non è l'unico modello possibile per rappresentare in modo generale una computazione: solo per citarne alcuni, ricordiamo il lambda-calcolo, le funzioni ricorsive, molti moderni linguaggi imperativi e funzionali...

Viene subito da chiedersi se, scegliendo un modello diverso, si potrebbero descrivere e quindi analizzare funzioni diverse, magari in numero maggiore rispetto a quelle che possono essere descritte e calcolate con la MdT.

La risposta a questo importante quesito venne data nei primi decenni del XX secolo. In questo periodo infatti si sono sviluppati i numerosi modelli computazionali, alcuni dei quali profondamente diversi dalla MdT, che cercavano di formalizzare la nozione di *algoritmo.*

Dobbiamo allora fare un passo indietro e pensare all'idea intuitiva che di esso ne abbiamo: un *procedimento* che consente di ottenere un *risultato*

atteso eseguendo, in un determinato ordine, un insieme di *passi semplici* corrispondenti ad azioni scelte solitamente da un *insieme finito*.

Definizione 1.3 (Funzione intuitivamente calcolabile)**.** Possiamo definire una funzione "intuitivamente" o "effettivamente" o "algoritmicamente" calcolabile quando esiste un *algoritmo* che la calcola.

Ora, si è dimostrato che i più svariati formalismi proposti sono tra loro equivalenti, e in particolare sono equivalenti alle MdT. Si ritiene dunque valida la

Tesi di Church-Turing La classe delle funzioni "intuitivamente" calcolabili coincide con quella delle funzioni calcolabili da una macchina di Turing.

In altre parole, questa *tesi* (che non è un teorema - in quanto fa riferimento a una nozione intuitiva come quella di algoritmo - ma solo una convinzione sul Mondo, per come lo comprendiamo al momento) asserisce che ogni dispositivo di calcolo fisicamente realizzabile (con silicio, DNA, neuroni o una tecnologia aliena) può essere simulato con una macchina di Turing.

Per il lettore che ha di certo familiarità con i linguaggi di programmazione tradizionali, non sarà difficile convincersi (e comunque può essere dimostrato formalmente) che un programma scritto in un linguaggio quale C o Java può essere scritto anche con una Macchina di Turing (e viceversa). Basterà infatti compilare il programma in un linguaggio macchina, che si presenta come una serie di istruzioni quali "leggi dalla memoria dei dati e salvali in un numero finito di registri" o "scrivi il contenuto di un registro in memoria" o "somma il contenuto di due registri". È facile immaginare di implementare queste operazioni con una MdT.

Visto quanto osservato, d'ora in poi eviteremo di descrivere gli algoritmi con il noioso formalismo delle MdT, tenendo però ben presente che sarebbe tecnicamente possibile!

1.3.3 Classi di Complessità

Possiamo ora definire formalmente cosa si intende per "misurazione delle risorse".

Per quel che riguarda la **misurazione della risorsa tempo**, data una macchina di Turing M, si dice che M opera in tempo $f(n)$ se, dato un input x di lunghezza n, la macchina M produce il risultato in $f(n)$ passi.

Per quel che riguarda la **misurazione della risorsa spazio**, data una macchina di Turing M, si dice che M opera in spazio $f(n)$ se, dato un input x di lunghezza n, la macchina M utilizza $f(n)$ celle "temporanee" per effettuare la computazione. Per *temporanee* si intende, come spiegheremo ampiamente in seguito, che la dimensione dell'input e la dimensione dell'output non vengono conteggiate in $f(n)$.

Partendo da queste misurazioni, sono state definite molte **classi di complessità.** Esaminiamone alcune:

- la classe $\mathbb{TIME}(f(n))$ è l'insieme dei problemi che ammettono una macchina di Turing che li risolve e che opera in tempo $f(n)$.
- la classe $\mathbb{NTIME}(f(n))$ è l'insieme dei problemi che ammettono una macchina di Turing non deterministica che li risolve e che opera in tempo $f(n)$.
- la classe $\mathbb{SPACE}(f(n))$ è l'insieme dei problemi che ammettono una macchina di Turing che li risolve e che opera in spazio $f(n)$.
- la classe $\mathbb{NSPACE}(f(n))$ è l'insieme dei problemi che ammettono una macchina di Turing non deterministica che li risolve e che opera in spazio $f(n)$.

È bene sottolineare ancora una volta come non sia determinante la scelta delle MdT come modello per la misurazione. Si ritiene infatti valida la

Tesi di Church-Turing "forte" Ogni modello computazionale fisicamente realizzabile può essere simulato da una MdT con un *overhead al più poli-*

nomiale per quanto riguarda il tempo, e un *overhead costante* per lo spazio. In altre parole, t passi nel modello scelto possono essere simulati con al più t^c passi di una MdT, e s celle di memoria del modello possono essere simulate con al più $d \cdot s$ celle della MdT, con c e d due costanti dipendenti dal modello.

1.3.4 Limiti in Spazio e Macchina di Turing Offline

Per studiare la complessità in spazio, è subito necessario chiedersi come calcolare l'utilizzo di spazio di una computazione. Inizialmente si potrebbe pensare che, avendo una MdT a k nastri, potremmo scegliere come misurazione per lo spazio utilizzato la *somma delle lunghezze delle stringhe* presenti su ciascun nastro alla fine della computazione (o anche durante, ma ipotizziamo che le stringhe non possano accorciarsi nel nostro modello).

Con questa visione, sarebbe impossibile considerare computazioni che utilizzino uno spazio *inferiore* a n, la lunghezza dell'input. Infatti, secondo il criterio ipotizzato in precedenza, la somma delle stringhe sarebbe *almeno* lunga n. Consideriamo però ora il seguente esempio:

Esempio Scriviamo una MdT con le seguenti caratteristiche: abbiamo a disposizione tre nastri ($k = 3$), in cui il primo contiene l'input e *non è mai sovrascritto*. La Macchina verifica se la stringa x è palindroma, implementando un ciclo da 1 a n, dove $n = |x|$. Per ogni i, la macchina verifica se $x[i] = x[n - i]$.

Il secondo nastro contiene il valore (in binario) del contatore del loop i, aumentato di 1 a ogni iterazione.

Il terzo nastro contiene il valore di un altro contatore, j, che serve a controllare i movimenti del cursore sul primo nastro. Ad ogni iterazione sostanzialmente la macchina raggiunge il simbolo $x[i]$, lo "memorizza" tramite l'automa a stati finiti, poi raggiunge $x[n - i]$ e lo confronta con il simbolo memorizzato: se sono uguali prosegue incrementando i, altrimenti risponde *no*.

Quanto spazio serve a questa macchina per operare? Di certo la macchina accede all'input, ma in modalità *read-only*. Le altre due stringe inoltre saranno lunghe (vista la rappresentazione binaria di i e j) al massimo $\log n$.

Possiamo dunque affermare che lo spazio utilizzato è $O(\log n)$.

Si può giungere allora alla seguente

Definizione 1.4 (Macchina di Turing Offline)**.** Una Macchina di Turing Offline (abbr. MTO) è una MdT a k nastri, con $k > 2$, in cui il **primo nastro** viene considerato come *input* e l'**ultimo** come *output*, e con due importanti restrizioni sulla funzione δ. Infatti se

$$\delta(q_i, a_1, ..., a_k) = (q_j, b_1, ..., b_k, m_1, ..., m_k)$$

allora deve valere che:

1. $a_1 = b_1$, ovvero che il carattere letto e scritto sul **primo nastro** sia lo stesso. In altre parole, il nastro che contiene l'input è *read only.*

2. $m_k \neq L$, ovvero che **l'ultimo nastro** (quello che contiene l'output) abbia una testina che si muove solo verso destra. In altre parole, tale nastro è *write only*

Si può facilmente intuire che questa definizione ci permette di *non considerare* i nastri di input e di output come spazio richiesto alla computazione.

Proposizione 1.3.1. *Una MTO ha lo stesso potere espressivo di una* MdT.

Dimostrazione. Sia S una MTO a $k+2$ nastri. Sia T una MdT a k nastri. Vogliamo simulare con S il funzionamento di T. S copia il suo input nel secondo nastro, e quindi simula il funzionamento di T nei suoi nastri da 2 a $k+1$. Quando T termina, S copia l'output nella stringa $k+2$, e poi termina.

□

Interazione tra MTO e l'ambiente

Da quanto detto, possiamo vedere la MTO come una macchina che non memorizza il suo input e il suo output (*e questo giustifica il fatto che non li consideriamo come spazio utilizzato*), bensì accede (random) a un input memorizzato esternamente e fornisce il suo output come uno stream di caratteri.

Se la tradizionale MdT viene vista come funzione $\Sigma^* \to \Sigma^*$, una MTO è invece una funzione che ha tipo

$$(State \times \Sigma) + \mathbb{N} \to (State \times \mathbb{N}) + \Sigma$$

dove con $+$ si vuole indicare l'unione disgiunta dei due insiemi[3].

La semantica della MTO può essere intesa nel modo seguente:

- se la macchina riceve dall'ambiente un input $n \in \mathbb{N}$, significa che le viene chiesto di calcolare l'n-simo carattere ($\in \Sigma$) del suo output
- se la macchina necessita invece di leggere un carattere del suo input, *dovrà richiederlo all'ambiente*, mettendo in output una coppia $\langle s, n\rangle \in State \times \mathbb{N}$ dove s è lo stato attuale della macchina (stato dell'automa, memory dump del nastro di lavoro - necessari a far ripartire in seguito la computazione), mentre n è l'indice del carattere che sta richiedendo. L'ambiente risponderà a questa richiesta facendo ripartire la macchina con input $\langle s,\, i_n\rangle \in State \times \Sigma$ dove i_n è proprio il carattere cercato.

Nonostante questo sembri un semplice tecnicismo, è il passo fondamentale per giungere alla **computazione bidirezionale.**

1.3.5 Esempi di Complessità in Spazio

Vediamo ora alcuni esempi, descritti in modo informale ma rigoroso, di problemi che presentano una diversa complessità in spazio (e le funzioni che essi rappresentano, dunque, appartengono a diverse classi di complessità).

[3]Questa unione dunque mantiene traccia dell'insieme di provenienza degli elementi, ovvero si definisce $A + B = \{inl(x)|x \in A\} \cup \{inr(y)|y \in B\}$.

Spazio Costante Consideriamo il problema PARITY: una stringa binaria appartiene al linguaggio se contiene un numero dispari di "uno".

Intuitivamente, la MdT richiede l'input un bit alla volta, e mantiene nel suo stato la parità dei bit letti fino a quel momento. Supponiamo per esempio che la macchina sia nello stato q_1 se i bit che valgono 1 letti fino a quel momento sono in numero dispari, e sia nello stato q_2 altrimenti. All'inizio la macchina sarà quindi nello stato q_2 (zero bit che valgono 1).

Quando la macchina legge uno 0, rimane nello stato precedente, mentre quando legge un 1, cambia stato. È immediato intuire come questo rifletta proprio la parità dei bit letti fino a quel momento.

Dunque la stringa verrà *accettata* se alla fine della lettura la MdT è nello stato q_1, verrà *rifiutata* altrimenti.

Lo spazio richiesto è costante, e in questo caso particolare è nullo, in quanto non abbiamo bisogno di nessun nastro di lavoro (l'unico ad incidere nel computo dello spazio occupato), infatti ci basta l'automa interno alla MdT per tenere traccia della "parità" della stringa.

D'altra parte ci si aspettava questo risultato, in quanto il linguaggio descritto può essere riconosciuto da un Automa a Stati Finiti.

Spazio Polinomiale Di questa classe (che chiameremo $\mathbb{PSPACE}$) possiamo dare un interessante esempio: il problema della *soddisfacibilità* (SAT).

Una **formula booleana** è un'espressione che coinvolge costanti (VERO e FALSO), variabili booleane e operatori booleani (AND, OR, NOT). Una formula è *soddisfacibile* se esiste almeno un assegnamento di VERO e FALSO alle variabili della formula che la rende *vera*.

Se, in termini di tempo, questo problema presenta serie difficoltà, per quanto riguarda lo spazio invece la situazione migliora, in quanto è possibile *riusare* lo spazio del nastro di lavoro.

Intuitivamente, possiamo descrivere una MdT che risolve SAT come segue: dato un *input* ϕ, dove ϕ è una formula booleana, *per ogni assegnamento di ve-*

rità alle variabili $x_1...x_m$ di ϕ, valuta $\phi(x_1...x_m)$. Se almeno un assegnamento valuta la formula a VERO, allora *accetta*, altrimenti *rifiuta*.

L'intuizione è la seguente: il nastro di lavoro può essere riutilizzato per ogni iterazione. La macchina infatti ha bisogno di memorizzare solo l'assegnamento di verità corrente, e questo può essere fatto in spazio $O(m)$. Ma il numero di variabili può essere *al più* uguale alla lunghezza della formula in input, che supponiamo essere n. Questo ci dice che l'algoritmo lavora in $O(n)$-*spazio*

Come generalizzazione del problema SAT possiamo introdurre il problema della *formula booleana quantificata* (QBF).

Vediamo la **formula booleana** già definita in precedenza in una forma più generale, ovvero con l'introduzione di due **quantificatori**: $\forall$ (*per ogni*) e $\exists$ (*esiste*). Scrivendo $\forall x\phi$ intendiamo che, *per ogni valore di x*, la formula ϕ è *vera*, mentre con $\exists x\phi$ intendiamo che esiste *almeno un valore di x* per il quale la formula ϕ è *vera*. La variabile che segue immediatamente il quantificatore è da esso **legata**.

Un quantificatore può apparire ovunque in una formula, in tal caso si applica al frammento di formula che sta dentro le parentesi che seguono il quantificatore. Tale frammento è lo **scope** del quantificatore.

Spesso è conveniente richiedere che tutti i quantificatori appaiano all'inizio della formula. In questo caso si dice che tale formula è in **forma normale prenessa**. La trasformazione è semplice, e dunque possiamo ipotizzare che i nostri enunciati siano in tale forma.

Quando ogni variabile in una formula si torva entro lo **scope** di un quantificatore, la formula si dice **chiusa** (si parla in questo caso di **enunciato**) e abbiamo la certezza che essa possa essere sempre e solo *vera* o *falsa.*

Il problema QBF dunque si riassume nel decidere se un ***enunciato*** è *vero* o no. Possiamo descrivere informalmente una macchina che riconosce se una formula ϕ è vera nel modo seguente:

- Se ϕ non contiene quantificatori, allora conterrà soltanto *costanti booleane,* per cui basterà valutarla per sapere se è *vera* o *falsa*.
- Se ϕ è nella forma $\exists x\psi$ chiamo ricorsivamente la macchina su ψ, prima assegnando a x il valore VERO, poi assegnandogli il valore FALSO. Se almeno uno dei due risultati è VERO, allora *accetta*, altrimenti *rifiuta*.
- Se ϕ è nella forma $\forall x\psi$ chiamo ricorsivamente la macchina su ψ, prima assegnando a x il valore VERO, poi assegnandogli il valore FALSO. Se entrambi i risultati valgono VERO, allora *accetta*, altrimenti *rifiuta*.

Per analizzare la complessità notiamo subito che la profondità massima delle ricorsioni coincide con il numero di variabili. A ogni livello, dobbiamo memorizzare unicamente il valore di una variabile, quindi la macchina lavora in spazio $O(m)$, dove m è il numero di variabili che appaiono in ϕ. Tale numero, come già visto, può essere *al più* n. Questo ci dice che l'algoritmo lavora in $O(n)$-*spazio*.

1.3.6 Complessità $\mathbb{LOGSPACE}$

La classe di complessità che ci interessa in questo lavoro è definita come $\mathbb{L} = \mathbb{LOGSPACE} = \mathbb{SPACE}(\log(n))$.

Esempio Un primo semplice esempio di problema risolubile in spazio logaritmico è il riconoscimento delle stringhe che appartengono al linguaggio $A = \{0^n 1^n | n \geq 0\}$.

Il funzionamento intuitivo ci dice che la macchina semplicemente scorrerà la stringa e memorizzerà separatamente due contatori, uno per contare gli *zeri*, uno per contare gli *uni*. Ovviamente, per un contatore grande al più n, in binario basterà una quantità di memoria pari a $log(n)$ per memorizzarlo. Dunque l'algoritmo esegue banalmente in $O(\log n)$-*spazio*.

Esempio Un altro interessante esempio è *l'algoritmo di Cook & McKenzie per testare l'aciclicità di un grafo indiretto*, che per altro viene implementato

in [DLS10b] con il linguaggio funzionale logspace che studieremo in questa tesi.

L'algoritmo può essere descritto utilizzando la nozione di *"cammino tenendo la destra":* il lettore immagini di camminare in un labirinto e di tenere sempre la sua mano destra sul muro. Nel nostro caso, gli archi saranno i corridoi, mentre i nodi rappresentano gli incroci del labirinto.

Si dimostra che un grafo è aciclico se e solo se ogni *cammino tenendo la destra* ritorna al punto di partenza attraversando l'arco che aveva usato, in direzione opposta, alla partenza.

Intuitivamente, per questo tipo di check, si devono tenere in memoria solo pochi (in numero costante rispetto all'input) nodi del grafo. Memorizzeremo allora dei puntatori a tali nodi: la dimensione di questi puntatori non è ovviamente costante, ma cresce in modo logaritmico (per indirizzare n nodi mi basterà infatti un puntatore lungo $\log n$).

Deduciamo quindi che lo spazio di lavoro necessario all'intero programma è *logaritmico* rispetto all'input.

1.4 Computazione Bidirezionale

Pensiamo alla **computazione bidirezionale** come una computazione *interattiva*: un input troppo grande per essere memorizzato non può, ovviamente, essere letto tutto in una volta; può altresì essere *richiesto pezzo per pezzo* durante l'esecuzione. Così *l'informazione non scorre semplicemente dall'input al programma,* bensì nella forma di richieste (*query*) anche *nella direzione opposta.*

Il modello della MTO descritto in precedenza cattura questo nuovo paradigma, che è dunque un buon modo di strutturare la computazione limitata in spazio.

Esempio: composizione di funzioni Supponiamo di avere due MTO, f e g, e di voler realizzare la composizione di queste due macchine, ovvero $f \circ g$

(dove $(f \circ g)(x)$ indicherà che vogliamo calcolare $g(f(x))$).

Non ci basterà semplicemente eseguire una macchina di seguito all'altra (anche perché questo potrebbe generare risultati intermedi che eccedono il limite di spazio logaritmico che vogliamo mantenere), dovremo invece lavorare con un flusso di dati bidirezionale. In particolare, possiamo pensare alla computazione come a un **dialogo** tra le due macchine: *ogni volta che l'ambiente richiede un bit di output alla composizione, la prima funzione richiede un output alla seconda macchina, la quale potrebbe necessitare, per il calcolo, di un carattere del suo input; poiché in questo caso l'input della seconda macchina coincide con l'output della prima macchina, quest'ultima sarebbe indotta ad iniziare una computazione.*

Caratteristiche È importante notare subito alcuni aspetti di questo tipo di computazione:

- l'ambiente può richiedere bit di output alla macchina, ma anche la macchina può richiedere bit di input all'ambiente;
- una macchina deve essere pronta a ricevere più volte la stessa richiesta e ricalcolare in ogni occasione il valore del bit domandato: questo ci appare sensato, visto che se memorizzassimo queste informazioni cadremmo ben presto nell'uso di spazio lineare rispetto all'input;
- in un sistema in cui la computazione avanza per continue richieste e risposte (*message passing*), anche la composizione di due funzioni totali potrebbe essere una funzione parziale.

Capitolo 2

Una Proposta: IntML

> *We have introduced [...] IntML with the intention of helping the programmer to implement functions with sublinear space usage.*
>
> – Ugo Dal Lago & Ulrich Schöpp

In questo capitolo verrà fornita un'analisi abbastanza approfondita di un linguaggio funzionale di recente introduzione che garantisce l'uso di spazio logaritmico rispetto all'input: **IntML**. A differenza degli articoli di ricerca in cui questo linguaggio nasce, che ne danno una definizione formale basata soprattutto sul lambda-calcolo, qui si cercherà di darne una definizione più "pratica". Verrà infatti data una descrizione dettagliata (un "manuale") della sintassi *concreta* che può essere utilizzata da un programmatore grazie all'interprete (per ora prototipale ma già dotato di molte caratteristiche utili) realizzato dagli stessi autori del linguaggio.

2.1 Un Linguaggio Per la Computazione Bidirezionale e Logspace

Ampi sono stati gli studi sugli algoritmi che utilizzano dati troppo grandi per essere contenuti in memoria. Al contrario, pochissimi si sono occupati di come programmare in modo naturale questi algoritmi.

Recentemente due articoli di ricerca [DLS10a, DLS10b] hanno introdotto un linguaggio che va in questa direzione: IntML.

Lo scopo di questo linguaggio è permettere la programmazione di algoritmi che utilizzano spazio logaritmico rispetto al loro input. Come spiegato in precedenza, questo è possibile, ad esempio, tramite il modello della Macchina di Turing Offline e della computazione bidirezionale. In particolare, il concetto nuovo e "sconosciuto" al programmatore di linguaggi tradizionali è quello di *computazione su richiesta*.

Per evitare i frequenti errori che si possono compiere lavorando con questo stile di programmazione, è necessario fornire al programmatore un valido supporto per utilizzarlo con semplicità.

A differenza di altri lavori, che nascondevano completamente l'implementazione della computazione su richiesta, IntML si propone di *arricchire* il linguaggio con delle *features utili* ad implementarla.

Nodi di Message Passing L'interpretazione che IntML dà della computazione bidirezionale vede una funzione come un *nodo* in una **rete** di *message passing*, in cui un nodo può avere uno o più *cavi* entranti e uscenti.

Ogni cavo può essere percorso in entrambe le direzioni da un *messaggio* (entrante o uscente). I nodi non hanno stato e reagiscono all'arrivo di un messaggio su un cavo ad essi connesso: quando viene recapitato loro un messaggio, lo usano come input per calcolare un output che viene passato ad un altro nodo (o all'ambiente) attraverso un (altro) cavo.

Per prima cosa, all'interno di ogni cavo che entra o esce da un nodo (Fig. 2.1) possiamo distinguere i due flussi (Fig. 2.2): uno nella direzione del cavo (che possiamo etichettare con un tipo[1] generico X^+) e uno in quella opposta (che etichetteremo con X^-).

Vediamolo con un esempio: vogliamo ora scrivere un nodo (una *funzione*) che rappresenti una parola binaria (w_1): interrogando la funzione dandole in input un indice, verrà restituito il valore del bit in quella posizione. Possiamo

[1] Dei tipi si parlerà fra poco, dopo la definizione del linguaggio.

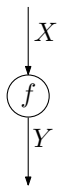

Figura 2.1

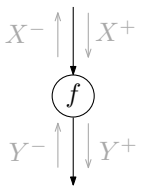

Figura 2.2

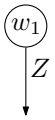

Figura 2.3

Figura 2.4

rappresentarlo con il nodo di Fig. 2.3: l'interfaccia che mette a disposizione è allora rappresentata da una coppia di tipi: (Z^-, Z^+), dove Z^- rappresenta le richieste inviate alla funzione (dovrà quindi essere un tipo che contiene abbastanza elementi per indirizzare tutti i bit della parola) e Z^+ rappresenta le possibili risposte (ovvero il valore del bit: 0 o 1).

Connettendo, per mezzo dei cavi, più nodi, si possono creare delle reti che rappresenteranno il nostro programma in IntML. In Fig. 2.4 si può vederne un semplice esempio: se connettiamo il nodo *get* al nodo w_1, allora il primo potrà interrogare il secondo inviandogli una richiesta (di tipo Z^-) nel verso opposto alla direzione del cavo, per poi ricevere, sullo stesso, la risposta (di tipo Z^+). Analogamente l'ambiente potrà inviare un messaggio di input (quindi di tipo U^-) a *get* per mezzo del cavo uscente, e aspettarsi un suo output (di tipo U^+) sullo stesso cavo.

Definizione del Linguaggio

Definizione 2.1 (IntML)**.** IntML è un linguaggio funzionale tipato con due classi di termini e tipi.

La definizione di IntML infatti inizia con un normale linguaggio funzionale, che andrà a costituire la **Working class** (WC) del linguaggio.

Per questa classe è stato scelto un semplice linguaggio del prim'ordine con tipi finiti (ma si potrebbe sceglierne uno più espressivo). Già con i tipi e i termini della WC è possibile implementare i circuiti (e quindi i

programmi bidirezionali), ma con molte difficoltà legate al nuovo stile di programmazione.

Si estende quindi questo linguaggio con delle primitive per costruire e manipolare le reti di message passing in modo *naturale*: chiamiamo questa l'**Upper class** (UC).

È bene sottolineare subito alcuni aspetti dell'UC.

- Le primitive dell'UC non rappresentano semplicemente i circuiti, ma si ispirano direttamente ai classici costrutti di un linguaggio di programmazione funzionale ad alto livello. È proprio questo che permetterà al programmatore di scrivere programmi bidirezionali senza cambiare il suo stile di programmazione.

- Le primitive sono *estensioni conservative* della WC, e dunque possono programmare solo circuiti che avremmo potuto scrivere, seppur con notevoli complicazioni, anche direttamente nella WC. Questo sarà evidente quando, parlando dell'interprete, mostreremo che i termini dell'UC, prima di essere valutati, vengono *compilati* in termini della WC.

Type Inference L'articolo [DLS10b] sviluppa un algoritmo di *type inference* per IntML, e questo rende molto più pratico il compito di programmare algoritmi $\mathbb{LOGSPACE}$ in IntML.

Infatti l'inferenza permette al programmatore di non curarsi dei bound relativi all'utilizzo di spazio (che diventano presto ingestibili), poiché essi vengono derivati dall'algoritmo.

Spazio Logaritmico Esiste una precisa corrispondenza tra la classe di funzioni rappresentabili in IntML e la classe $\mathbb{LOGSPACE}$ (cioè delle funzioni calcolabili in spazio logaritmico). In particolare, si dimostrano i seguenti:

Teorema 2.1.1 (Correttezza logspace)**.** *Sia* `t` *un termine scritto in IntML e sia* $\phi : \{0,1\}^* \to \{0,1\}^*$ *la funzione che esso rappresenta, allora vale che* ϕ *è calcolabile in spazio logaritmico.*

Inoltre, la valutazione in IntML di `t` *fornisce un algoritmo logspace per calcolare* ϕ*.*

Teorema 2.1.2 (Completezza logspace)**.** *Qualsiasi funzione* $\phi : \{0,1\}^* \to \{0,1\}^*$*, calcolabile in spazio logaritmico, può essere rappresentata con un certo termine* `t` *appartenente all'UC di IntML.*

Per la dimostrazione del Teorema 2.1.1 si fa riferimento alla traduzione di `t` in un circuito scritto nella WC, mostrando come la computazione di tale circuito richieda spazio logaritmico.

La dimostrazione del Teorema 2.1.2 invece si ottiene mostrando che IntML può codificare la funzione δ di una Macchina di Turing Offline che calcola la funzione ϕ.

Entrambe sono fornite nel dettaglio nel paragrafo 5 di [DLS10a].

2.2 L'Interprete

Per il linguaggio è disponibile un interprete (ancora in fase prototipale) che permette, tramite un'interfaccia a caratteri, di fare parsing di espressioni scritte in IntML e in seguito di valutarle.

Valutazione Se la valutazione di un termine WC può avvenire direttamente, la valutazione di un termine UC avviene in due fasi: prima si *compila* (o, per essere più precisi, si *traduce*) in un termine equivalente della WC, e poi si valuta quest'ultimo.

Per lanciare la valutazione di un termine della WC basterà scrivere il suo nome dopo il prompt #. Lo stesso non vale per i termini dell'UC: se `term` è un termine UC, allora per lanciare la sua valutazione bisognerà digitare, dopo il prompt, `term <>`.

Nomi L'interprete ci dà la possibilità di assegnare dei *nomi* ai termini, così da renderne più semplice la valutazione e il riuso, tramite gli operatori

=W e =U, da utilizzare rispettivamente per i termini della Working e quelli dell'Upper Class.

Annotazione di Tipo Per tutti i termini dell'UC e per la costante min della WC è possibile aggiungere un'annotazione di tipo, per forzare l'algoritmo di type inference ad utilizzare quel tipo più specifico per quel termine. L'annotazione di tipo si indica con i due punti (:), seguiti dalla sintassi concreta per i tipi, che verrà spiegata in seguito.

```
1 t =W min : 1+(1+(1+1));
2
3 y =U (fun f : {'e}({'c}'a --o 'b) --o 'd -> f (fun x -> x));
```

Commenti Possono essere inseriti commenti racchiusi tra (*..*):

```
1 (*sono solo un commento... ma sono comunque molto utile!*)
```

2.3 Working Class

Negli articoli citati, è stato scelto come WC un semplice linguaggio funzionale al prim'ordine con tipi finiti.

2.3.1 Sistema di Tipi

Il sistema di tipi della WC è formato da tipi al prim'ordine con variabili di tipo. In BNF, possiamo definirli così:

$$A,\ B ::= \texttt{'a} \mid \texttt{1} \mid A{*}B \mid A{+}B$$

dove :

- 'a (e 'b,' c, ...) sono variabili di tipo, cioè tipi generici non ancora istanziati, che potranno assumere una qualche composizione legale di altri tipi.

- `1` rappresenta il tipo del *singoletto*, ovvero un tipo (inteso come *insieme di valori possibili*) che contiene un solo elemento. La cardinalità di tale insieme sarà ovviamente $|\texttt{1}| = 1$.

- Il costruttore $+$ serve a costruire l'unione disgiunta dei tipi A e B. Ricordiamo che questo tipo di unione mantiene traccia dell'insieme di provenienza degli elementi. La cardinalità dell'insieme sarà $|A+B| = |A| + |B|$.

- Il costruttore $*$ serve a costruire il prodotto cartesiano tra A e B. Gli elementi di tale insieme saranno *coppie* il cui primo elemento appartiene ad A e il secondo appartiene a B. La cardinalità dell'insieme sarà $|A*B| = |A| * |B|$.

Da queste considerazioni, è facile evincere che, per tutti i tipi, *l'insieme dei valori possibili è finito*. Possiamo infatti vedere $+$ e $*$ come operatori sulla cardinalità dei tipi.

2.3.2 Termini

Per la costruzione di programmi in IntML vogliamo denotare e manipolare dei valori. Lo possiamo fare utilizzando i termini che la WC ci mette a disposizione.

Singoletto Il singoletto è l'unico valore presente nel tipo `1` e si denota con <>. Ad esempio, si può scrivere:

```
1 singoletto =W <>;
```

Iniezione Sinistra e Destra Per denotare gli elementi di tipo $A + B$, mantenendo però l'informazione sull'insieme originario di quell'elemento, si possono utilizzare le *iniezioni sinistra e destra*.

Possiamo dunque scrivere `inl(f)` oppure `inr(f)` dove `f` è un termine.

True e False Il tipo dei booleani è, come prevedibile, definito come $1 + 1$. Ne segue immediatamente che si possono dare le seguenti definizioni:

```
1 true =W inl(<>);
2 false =W inr(<>);
```

Ordine Totale sui Tipi Per ogni tipo A sono presenti tre costanti: **min**, **succ** ed **eq**, che forniscono su quel tipo una *relazione d'ordine totale* (quindi è sempre possibile ordinare e confrontare elementi di un certo tipo).

Come è naturale pensare, si possono scrivere (e li vedremo in alcuni esempi seguenti) dei termini quali `pred` e `max`, di ovvia semantica.

Questo ci permette, ad esempio, di considerare un generico tipo come un segmento iniziale dell'insieme dei *naturali:*

```
1 zero =W min : 1+(1+(1+1));  (*in questo modo istanzio il tipo
      di cui mi sto occupando a contenere 4 elementi (0..3)*)
2 uno =W succ zero;
3
4 uguali =W (eq zero min);
5 uguali2 =W zero = min;  (*che e' solo zucchero sintattico
      rispetto al termine precedente*)
6
7 diversi =W (eq uno zero);
```

Per capire meglio quanto visto, presentiamo l'output della valutazione di tali termini da parte dell'interprete:

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

# uno
 : 1 + (1 + (1 + 1))
= inr(inl(<>))

# uguali
 : 1 + 1
```

```
= inl(<>)

# uguali2
 : 1 + 1
= inl(<>)

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

Costruttore di Coppie Per costruire una *coppia*, ovvero un elemento che appartiene al *prodotto cartesiano*, si utilizza il costrutto `<f,g>`, dove `f` e `g` sono dei termini della WC. Ad esempio:

```
1 f =W ...;
2 g =W ...;
3 coppia =W <f, g>;
```

Distruttori di Coppie Per estrarre i valori da una coppia non esistono costrutti nativi, ma essi sono facilmente definibili tramite **`fun`** e **`let`**, di cui parleremo estesamente nella descrizione dell'Upper Class. Quindi si può scrivere:

```
1 ...
2 fst =W fun x -> let x be <x1, x2> in x1;
3 snd =W fun x -> let x be <x1, x2> in x2;
4
5 primo =W fst coppia;
6 secondo =W snd coppia;
```

Intuitivamente, `fst` prende in input un termine `x`, lo riduce fino a che non è nella forma `<v, w>` e a questo punto sostituisce `v` a `x1` e `w` a `x2` nel termine seguente (che in questo caso è proprio `x1`).

Case Il linguaggio ci mette a disposizione un costrutto di *case-distinction* basato sull'unione disgiunta.

Il costrutto si presenta nella forma

case f **of** inl(c) -> g | inr(d) -> h

Moralmente, si valuta f finché non raggiunge la forma inl(v) o inr(v). A questo punto si esegue la distinzione e si restituisce g o h in cui però si sostituisce v a c o d, dove c e d sono variabili libere che compaiono rispettivamente in g e in h. Più formalmente, presentiamo la *regola di riduzione (riscrittura)* (indicata con $\longrightarrow$) che governa questo costrutto:

$$\texttt{case}\, inl(v)\, \texttt{of}\, inl(c) \Rightarrow g \,|\, inr(d) \Rightarrow h \quad \longrightarrow \quad g\{v/c\}$$
$$\texttt{case}\, inr(v)\, \texttt{of}\, inl(c) \Rightarrow g \,|\, inr(d) \Rightarrow h \quad \longrightarrow \quad h\{v/d\}$$

Possiamo quindi vedere un esempio di funzionamento:

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

La valutazione di t ci restituirà (in questo caso) il primo elemento della coppia se h =W inl(coppia) e il secondo elemento se invece h =W inr(coppia).

If-then-else Ci viene messo a disposizione il costrutto

if condizione **then** t_se **else** t_altrimenti

con il funzionamento atteso: si valuta "condizione", se si riduce a inl(<>) (cioè true) allora il termine if assume il valore di "t_se", se invece si riduce a inr(<>) (cioè false) allora assume il valore di "t_altrimenti".

Questo ci permette di scrivere facilmente alcuni operatori booleani. Il fatto di *non poter utilizzare il solo ramo* ***then*** permette di evitare qualsiasi ambiguità di annidamento dei costrutti **if**.

```
and =W fun x -> fun y ->
  if x then
```

```
3     if y then
4       true
5     else
6       false
7   else
8     false;
9
10 or =W fun x -> fun y ->
11   if x then
12     true
13   else
14     y;
```

Ciclo Per realizzare un loop abbiamo a disposizione il costrutto `iter`, che si presenta nella forma

`(iter c -> f) g`

La semantica operazionale intesa può essere chiarita in un qualche pseudolinguaggio:

```
h=f[g/c];  //c è una variabile libera dentro f
while h=inr(j) do
   h=f[j/c];
// se sono uscito dal ciclo significa che h è
// nella forma inl(i), restituisco i.
let h=inl(i) in i;
```

All'interno di `f` dunque restituiremo `inr(j)` se vogliamo che l'iterazione prosegua sul termine `j`, mentre `inl(i)` se vogliamo che l'iteratore restituisca `i`.

Un interessante uso di **`iter`** può essere la definizione dei termini `pred` e `max`, che operano su un qualsiasi tipo A:

```
1 max =W
2  (iter z -> if succ z = z then inl(z) else inr(succ z)) min;
3
4 pred =W fun x ->
```

```
5          if x = min then min else
6            ((iter y -> if (succ y) = x then inl(y) else
                inr(succ y)) min);
```

2.4 Upper Class

L'UC fornisce i costrutti per la programmazione in spazio limitato.

2.4.1 Sistema di Tipi

Il sistema di tipi dell'UC può essere espresso in BNF:

$$X, Y ::= [A] \mid X * Y \mid \{A\}X - \circ Y$$

in cui X e Y sono tipi dell'UC, mentre A è un qualsiasi tipo della WC. Analizziamoli nel dettaglio.

- $[A]$ mi permette di utilizzare nell'UC il tipo della WC A, opportunamente "confezionato" in un **box**.

- $X * Y$ è sostanzialmente il *prodotto cartesiano* tra i due tipi: serve a definire le *coppie*. Formalmente è definito con l'operatore di *tensore*: $X \otimes Y$.

- $\{A\}X - \circ Y$ vuole rappresentare lo spazio di funzioni indicizzato (che formalmente potremmo scrivere come $A \cdot X \multimap Y$). Come sappiamo, i linguaggi funzionali permettono di avere in input e in output delle *funzioni*. Questo rappresenta proprio il tipo delle *funzioni che prendono in input un elemento di X e restituiscono un elemento di Y*.
  In IntML bisogna però tener conto di una complicazione: per tenere la *complessità in termini di spazio* bassa, dobbiamo indicare esplicitamente *quante volte, al più, posso usare l'argomento in input*. Per fare questo indicizzo X con un tipo A della Working Class.
  Si può ad esempio scrivere $2 \cdot [2] \multimap [2]$ (ipotizzando di chiamare per

semplicità 2 il tipo $1 + 1$), che rappresenta l*e funzioni che prendono in input un booleano, lo utilizzano al più due volte e restituiscono un booleano.*

Se non viene indicato nulla, A viene istanziato ad 1 e quindi si potrà utilizzare quell'argomento una sola volta.

2.4.2 Termini

Box La WC rimane qui totalmente disponibile. Sia `f` un termine della WC, allora potrò utilizzarlo tramite il termine `[f]` dell'UC.

```
1 f =W ...;
2 ...
3 boxf =U [f];
```

Distruttore del Box Si può avere la necessità di utilizzare il contenuto di un box, per esempio in un altro termine della WC. Si può in questo caso utilizzare il costrutto

let s **be** [c] **in** t

che opera nel modo seguente: riduce `s` fino a portarlo nella forma `[v]` e poi sostituisce il valore di `v` a `c`, che si suppone essere una variabile libera in `t`.

Vediamolo in un esempio: supponiamo di avere un termine `addW` che somma due tipi generici della WC, e supponiamo ora di volerlo utilizzare nell'UC.

```
1 (*add between two generic types 'a and 'b*)
2 addW =W ...;
3
4 add =U fun x -> fun y ->
5    let x be [xc] in
6    let y be [yc] in
7    [addW xc yc];
```

Costruttore di Coppie Esattamente come per la WC, anche qui si utilizza il costrutto `<s,t>`, dove `s` e `t` sono dei termini della UC.

Distruttore di Coppie Per estrarre i valori da una coppia ci viene messo a disposizione il costrutto

let s **be** <x,y> **in** t

che opera nel modo seguente: riduce `s` fino a portarlo nella forma `<v,w>` e poi sostituisce il valore di `v` e di `w` rispettivamente a `x` e a `y`, che si suppone siano due variabili libere che compiano *una sola volta* in `t`.

Possiamo ad esempio scrivere le proiezioni sulla coppia, per estrarne le componenti:

```
1 fstU =U fun x -> let x be <x1, x2> in x1;
2 sndU =U fun x -> let x be <x1, x2> in x2;
```

Lambda Astrazione Per costruire delle funzioni, abbiamo bisogno di indicare i parametri in input per quella funzione. Possiamo farlo con il costrutto

fun x -> t

in cui `x` deve essere una variabile libera che *compare una sola volta in* `t`.

Vediamo un semplice esempio, costruendo una funzione con due argomenti che rappresentano due bit, e ne esegue il prodotto.

```
1 zero =W inr(<>);
2 uno =W inl(<>);
3 ...
4 binmult =U fun x -> fun y ->
5   let x be [a] in
6   let y be [b] in
7     if (a = uno) then
8       if (b = uno) then [uno] else [zero]
9     else [zero];
```

Applicazione Per poter utilizzare le funzioni, è necessario applicarle a dei termini. In particolare possiamo scrivere l'applicazione di un termine a un altro come

```
s t
```

dove il primo si intende come funzione e il secondo come argomento. È naturale generalizzare questo costrutto a n argomenti.

Vediamo ad esempio come applicare la funzione per la moltiplicazione binaria scritta in precedenza

```
test_mult =U binmult [zero] [uno];
test_mult2 =U binmult [uno] [uno];
```

Valutando i due termini di test, otteniamo i risultati attesi.

Utilizzo Ripetuto di Argomenti I costrutti `let` e `fun` ci permettono di usare la variabile libera una sola volta. Possiamo però duplicare una variabile un numero finito di volte con il costrutto

```
copy x as x1,x2 in t
```

che sostanzialmente valuta `x` e sostituisce tale valore a `x1` e `x2`, che si suppone siano due variabili libere che compaiono *una sola volta* in `t`.

Ad esempio, se in un termine prevediamo di dover usare 5 volte una variabile, possiamo scrivere all'inizio della definizione di tale termine:

```
multiplevar =U
  fun x ->
    copy x as x1,x2345 in
    copy x2345 as x2,x345 in
    copy x345 as x3,x45 in
    copy x45 as x4,x5 in
    ...
    ...
```

Case Anche nella UC è presente un costrutto di *case-distinction*, che non è altro che una generalizzazione di quello della WC. Infatti si presenta nella consueta forma

```
case f of inl(c) -> x | inr(d) -> y
```

dove `f`, `inl(c)` e `inr(d)` sono termini della WC, mentre `x` e `y` sono termini dell'UC.

Possiamo servircene ad esempio per costruire un nuovo *if*, in cui la condizione appartiene alla WC, mentre i termini restituiti nell'uno o nell'altro caso sono della UC:

```
1 ifU =U fun x -> fun y -> fun z->
2   let x be [c] in
3     case c of
4       inl(d) -> y
5     | inr(d) -> z;
```

Hack È lo strumento per scrivere nell'UC tutti quei *circuiti* che non possono essere realizzati usando i costrutti predefiniti.

Il costrutto **hack** (che funziona un po' come l'*assembly inline* in C) permette infatti di descrivere, caso per caso, le azioni che un circuito compie a seconda degli input che riceve. È sostanzialmente un modo per implementare direttamente (usando quindi la WC) i circuiti, senza essere vincolati al limitante uso di *box*.

La sintassi presenta un termine che prende in input un parametro ed effettua una case distinction su di esso, per decidere il valore da restituire o le azioni da compiere (scritte con i costrutti della WC), salvo poi indicare che tipo dell'UC si vuole assegnare al termine creato.

Ricordando quanto accennato precedentemente sui circuiti, è bene evidenziare che la funzione che diventa argomento di **hack** prende un parametro di tipo X^- e restituisce un oggetto di tipo X^+, dove X è il tipo che vogliamo dare all'intero termine costruito.

```
loop =U hack fun x ->
  case x of
    ...
as {'a}({'c}['a] --o ['a + 'b]) --o ({'d}['a] --o ['b]);
```

Nel paragrafo seguente vedremo un esempio di utilizzo di **hack**.

2.5 Utili Esempi

Diamo alcuni esempi concreti di utili funzioni che possono essere scritte facilmente in IntML.

Parole Binarie Essendo in ottica funzionale, anche una stringa di n bit deve essere vista ovviamente come una funzione. In particolare, potrebbe essere una funzione con tipo

$$[\alpha] \multimap [2]$$

dove α rappresenta la lunghezza della stringa. La funzione può essere interrogata, per chiedere il valore dell'i-esimo bit, che può essere ovviamente visto come un booleano (quindi di tipo $1+1$). Possiamo ad esempio definire alcune stringhe di bit nel modo seguente:

```
(*La stringa 0..0111 *)
w1w =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;

w1 =U fun x -> let x be [c] in [w1w c];


(* La stringa 101 *)
w2 =U
  fun x ->
     let x be [c] in
```

```
15              case c of
16              inl(u) -> [uno]
17             |inr(d_t) -> (
18                case d_t of
19                inl(d) -> [zero]
20               |inr(t) -> [uno]);
```

Loop Possiamo definire un loop della UC con il seguente tipo (semplificato dalle annotazioni sullo spazio da utilizzare)

$$[\alpha] \multimap [\alpha + \beta] \multimap [\alpha] \multimap [\beta]$$

il cui funzionamento è il seguente:

$$\texttt{loop f v} = \begin{cases} \texttt{loop f [w]} & \textit{se } \texttt{f v = [inl(w)]} \\ \texttt{[w]} & \textit{se } \texttt{f v = [inr(w)]} \end{cases}$$

Con l'utilizzo del particolare costrutto **hack** possiamo definire loop nel modo seguente:

```
1  loop =U hack fun x ->
2    case x of
3      inl(y) ->
4        let y be <store, stepq> in
5          case stepq of
6            inl(argq) ->
7             let argq be <argstore, unit> in
8               inl(<store, inl(<argstore, store>)>)
9          | inr(continueOrStop) ->
10            case continueOrStop of
11              inl(continue) -> inl(<continue, inr(<>)>)
12            | inr(stop) -> inr(inr(stop))
13     | inr(z) ->
14       case z of
15         inl(basea) ->
16            let basea be <junk, basea> in
17              inl(<basea, inr(<>)>)
```

```
18      | inr(initialq) ->
19          inr(inl(<<>, <>>))
20   as {'a}({'c}['a] --o ['a + 'b]) --o ({'d}['a] --o ['b]);
21
22  continue =W fun x -> inl(x);
23  return =W fun x -> inr(x);
24  (* inl significa "continua con il ciclo", inr significa invece
       "fermati e ritorna questo valore" *)
```

Grafi Anche dei grafi bisognerà dare una rappresentazione funzionale *all'ordine superiore*. Essi possono essere visti come una coppia di predicati: il primo prende in input un elemento di un certo tipo e ci dice se quell'elemento appartiene o no al grafo, mentre il secondo prende in input una coppia di nodi e ci dice se esiste un arco che lega quei nodi. Più formalmente, potremmo descriverli con un tipo

$$([\alpha] \multimap [2]) \otimes ([\alpha \times \alpha] \multimap [2])$$

Ad esempio possiamo scrivere un grafo e alcune semplicissime funzioni per accedervi:

```
1  (* A graph is a pair of predicates (['a]--o[2]) *
       (['a*'a]--o[2]).
2   * We use the following functions to access the two
       components.*)
3  node =U fun graph -> pr1 graph;
4  edge =U fun graph -> pr2 graph;
5
6  (* Source and destination of an edge *)
7  src =W fun edge -> pi1 edge;
8  dst =W fun edge -> pi2 edge;
9
10 examplegraph =U
11   <fun x  ->
12      let x be [c] in
13               case c of
14               inl(d) -> [true]
```

```
15              |inr(d) -> [true],
16    fun x ->
17      let x be [c] in
18              [let c be <c1,c2> in
19              case c1 of
20                          inl(d) ->
21                               (case c2 of
22                                 inl(e) -> true
23                               | inr(e) -> true)
24                     |inr (d) ->
25                               (case c2 of
26                                 inl(e) -> true
27                               | inr(e) -> true)]>;
```

Capitolo 3

Programmi Logspace per Funzioni Aritmetiche

"Arithmetic is being able to count up to twenty without taking off your shoes"
– Anonymous

In questo capitolo cercheremo di realizzare, utilizzando IntML, una libreria di funzioni aritmetiche che utilizzano spazio di calcolo logaritmico rispetto all'input.

È interessante notare come i metodi (gli algoritmi) che ci vengono insegnati alle scuole elementari utilizzino spesso spazio lineare o addirittura quadratico rispetto all'input. Anche gli algoritmi proposti da testi matematici o informatici si preoccupano sostanzialmente solo della *complessità in termini di tempo*, e pochi si interessano a limitare l'uso dello *spazio*.

3.1 Div e Mod

Siano a e b due numeri naturali, con $b \neq 0$, allora *esistono e sono unici* due numeri naturali q ed r tali che: $a = qb + r$ con $0 \leq r < b$.

Questo ci permette di definire la *divisione intera* tra a e b, il cui quoziente è q: come si usa scrivere in Informatica, `a div b = q`. Analogamente, sappiamo che r sarà il resto di tale divisione, e dunque scriveremo `a mod b = r`.

Poiché stiamo lavorando con dei numeri naturali, la definizione di `div` equivale a quella di `floor`, ovvero la divisione tra a e b arrotondata all'intero inferiore più vicino.

3.1.1 L'Algoritmo

Come già visto, IntML ci permette di vedere tutti i tipi come un ordine totale, e quindi identificarli per esempio con i numeri naturali. Vogliamo scrivere una funzione `divmod` che calcola *il quoziente e il resto della divisione tra un termine* `x` *di tipo* `'a` *e la costante* `2`*.*

L'algoritmo è molto semplice: si parte da `x` e gli si sottrae `2` ripetutamente, finché non rimane *zero* o *uno*. Il valore finale sarà il resto della divisione (`mod`), mentre il numero di sottrazioni fatte, mantenuto in un apposito contatore, sarà il valore della divisione intera (`div`). Questi due valori vengono restituiti in una coppia.

Formalmente, la funzione avrà tipo

```
divmod :U ['a] --o ['a*'a]
```

Sarà poi semplice scrivere due funzioni che prelevano i singoli valori: `floor` e `mod2`. Per quest'ultima utilizzeremo un piccolo accorgimento per fare in modo che il risultato venga correttamente tipato come un booleano (il resto della divisione per 2 vale ovviamente 0 o 1).

3.1.2 Sorgente

Possiamo scrivere in IntML le funzioni descritte in precedenza.

```
1 (*Computes integer division by 2 and its mod*)
2 divmod =U
```

```
3   fun x : ['a] ->
4     let x be [xw] in
5     loop
6     (
7       fun v ->
8        let v be [vw] in
9     (*(fst vw) will count iterations, i.e. the result of
          integer division. (snd vw) holds the partial results of
          subtraction and will become the mod *)
10         case (or ((snd vw) = min : 'a) ((snd vw) = (succ min :
               'a)) ) of
11              inl(true) -> [return(vw)]
12             |inr(false) -> [continue(<(succ (fst vw)), (pred
                   (pred (snd vw)))>)] (* < d++, m--2 >*)
13    )
14    ([< min : 'a, xw>] : ['a * 'a]); (*the first value will
          count iterations, i.e. the result of integer division.
          the second value holds the partial results of
          subtraction *)
15
16 floor =U fun x -> let (divmod x) be [c] in [fst c];
17 mod2  =U fun x -> let (divmod x) be [c] in [if ((snd c) = min :
     'a) then zero else uno ];
```

3.2 Confronto

Gli algoritmi che presenteremo in seguito prendono in input *parole binarie*, codificate come spiegato nella Sezione 2.5.

Questo primo algoritmo prende in input due parole binarie e le confronta, restituendoci un valore che indica se la prima è maggiore, minore o uguale della seconda.

3.2.1 L'Algoritmo

L'idea è la seguente: si confrontano le parole binarie bit a bit, partendo dal bit più significativo fino a quello meno significativo. Finché i bit delle due parole sono uguali, si prosegue. Appena si incontrano due bit diversi, il risultato del confronto tra le parole sarà lo stesso del confronto tra quei due bit. Se si arriva agli ultimi due bit, ugualmente il risultato sarà semplicemente il confronto tra quei due bit.

Formalmente, la funzione `cmpL` avrà tipo

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

dove il tipo del risultato contiene appunto 3 valori: `inl(<>)` per indicare che le stringhe sono uguali, `inr(inl(<>))` per indicare che la prima è minore della seconda e infine `inr(inr(<>))` per indicare che la prima è maggiore.

3.2.2 Sorgente Completo

Possiamo scrivere in IntML la funzione `cmpL`, avvalendoci di funzioni ausiliarie già descritte in precedenza.

```
1  (*========== UTILS ==========*)
2
3  true =W inl(<>);
4  false =W inr(<>);
5
6  zero =W min : 1+1;
7  uno =W succ min :1+1;
8
9
10 and =W fun x -> fun y ->
11   if x then if y then true else false else false;
12 or =W fun x -> fun y ->
13   if x then true else y;
14 not =W fun x -> if x then false else true;
15
```

```
(* 'a maximum *)
max =W
 (iter z -> if succ z = z then inl(z) else inr(succ z)) min :
    'a;

(* 'a predecessor *)
pred =W fun x ->
         if x = min : 'a then min : 'a else
           ((iter y -> if (succ y) = x then inl(y) else
              inr(succ y)) min : 'a);

loop =U hack fun x ->
  case x of
    inl(y) ->
      let y be <store, stepq> in
        case stepq of
          inl(argq) ->
           let argq be <argstore, unit> in
             inl(<store, inl(<argstore, store>)>)
        | inr(continueOrStop) ->
           case continueOrStop of
             inl(continue) -> inl(<continue, inr(<>)>)
           | inr(stop) -> inr(inr(stop))
  | inr(z) ->
    case z of
      inl(basea) ->
          let basea be <junk, basea> in
            inl(<basea, inr(<>)>)
    | inr(initialq) ->
        inr(inl(<<>, <>>))
 as {'a}({'c}['a] --o ['a + 'b]) --o ({'d}['a] --o ['b]);
continue =W fun x -> inl(x);
return =W fun x -> inr(x);
(* left means continue with the loop, right means stop with
   this value *)

(*Upper class if*)
ifU =U fun x -> fun y -> fun z->
```

```
51   let x be [c] in
52     case c of
53       inl(d) -> y
54     | inr(d) -> z;
55
56 (*========== SUBROUTINES ==========*)
57
58 (*compare results: I use a type 1+(1+1) to cover all cases*)
59 res_eq =W inl(<>);
60 res_lt =W inr(inl(<>));
61 res_gt =W inr(inr(<>));
62
63 (*compare 2 bits and return type 1+(1+1) to cover all cases*)
64 bitcmp =W fun x -> fun y ->
65   if (x = zero) then
66     (if (y = zero) then res_eq else res_lt)
67   else (*x=uno*)
68     (if (y = zero) then res_gt else res_eq);
69
70
71 (*========== ALGORITHM ==========*)
72
73 (*I compare bit by bit, from msb. I stop when I find two
       different bits or reach the lsb
74 The result of the compare will be the result of the comparison
       of these bits.*)
75
76 cmpL =U (* cmpL: (['a] --o [2]) --o (['a] --o [2]) --o [3]*)
77  fun n1 ->
78  fun n2 ->
79       loop (* compare from msb of binary words and stop when I
              find a difference*)
80       (
81           fun v ->
82             let v be [vw] in
83             let (n1 [vw]) be [b1] in
84             let (n2 [vw]) be [b2] in
85               (ifU ([or (not ( (bitcmp b1 b2) = res_eq)) (vw =
```

```
                    min :'a)])  (*return the compare between two
                     bit if it is different or if they are the last
                     ones*)
86                  ([return(bitcmp b1 b2)])
87                  ([continue(pred vw)])
88                )
89        )
90        [max]; (*max will indicate the msb *)
91
92
93  (*========== TESTS ==========*)
94
95  w1 =U
96    fun x->
97       let x be [c] in
98                case c of
99                inl(u) -> [uno]
100              |inr(d_t) -> (
101                 case d_t of
102                 inl(d) -> [uno]
103                |inr(t) -> [uno])
104               ;
105
106 w2 =U
107   fun x ->
108      let x be [c] in
109               case c of
110               inl(u) -> [uno]
111              |inr(d_t) -> (
112                 case d_t of
113                 inl(d) -> [uno]
114                |inr(t) -> [zero]);
115
116 testgt =U cmpL w1 w2;
117 testlt =U cmpL w2 w1;
118 testeq =U cmpL w1 w1;
119 testeq2 =U cmpL w2 w2;
```

3.3 Addizione

Vogliamo ora realizzare l'addizione tra due *parole binarie*.

L'idea iniziale è quella che ci aspettiamo: utilizzare l'algoritmo imparato alla scuola elementare, sommando bit a bit a partire da quello meno significativo. Subito però ci rendiamo conto che per rimanere in spazio logaritmico non possiamo mantenere in memoria l'input e l'output completi, bensì dobbiamo codificare tanto gli operandi in ingresso quanto l'output con delle funzioni. Queste funzioni potranno essere interrogate tramite un indice, per richiedere uno specifico bit degli addendi o della somma risultante.

3.3.1 Implementazione

Definiamo per prima cosa alcuni utili operatori booleani:

```
1 and =W fun x -> fun y ->
2   if x then if y then true else false else false;
3
4 or =W fun x -> fun y ->
5   if x then true else y;
6
7 not =W fun x -> if x then false else true;
8
9 xor =W fun x -> fun y -> (not (x = y));
```

Abbiamo poi bisogno di uno strumento per sommare due bit, tenendo traccia però anche dei riporti in ingresso e in uscita. Utilizzeremo allora un **full-adder**.

Definizione 3.1 (full-adder)**.** Un **full-adder** è una rete combinatoria con tre ingressi e due uscite. La prima uscita è valorizzata con la *somma* dei bit in ingresso (che sono tre: *due operandi* e un *riporto in entrata*), mentre la seconda è valorizzata con il *riporto in uscita* generato dalla somma dei bit in ingresso.

Con gli operatori introdotti in precedenza, è facile implementarlo in IntML:

```
(*Full adder: performs an addition operation on bit (two digits
    and a previous carry)
and produces a two-bit output (the sum plus the carry)*)
fa =W fun x -> fun y -> fun c ->
   <xor x (xor y c),
    or (and x y) (or (and y c) (and x c))>;
```

Possiamo a questo punto realizzare la funzione `addL`. Essa prenderà in input due parole binarie e restituirà una parola binaria, codificata come funzione. Formalmente l'addizione avrà tipo

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

Quando si richiede un bit del risultato, la funzione in output lo calcolerà nel seguente modo: tramite un ciclo essa comincia a sommare i bit delle due parole partendo dalla coppia di bit meno significativi, tramite il full-adder, passando poi il valore del riporto all'iterazione successiva. Poiché non possiamo memorizzare i bit delle colonne precedenti (altrimenti cadremmo presto in un uso *lineare* dello spazio), ogni volta che viene richiesto un bit bisogna ricalcolare di nuovo tutti i bit del risultato precedenti.

Quando si giunge alla colonna che corrisponde al bit del risultato cercato, esso verrà calcolato - sempre tramite l'uso del full-adder - e restituito.

3.3.2 Sorgente Completo

```
(*========== UTILS ==========*)

true =W inl(<>);
false =W inr(<>);

uno =W true;
zero =W false;

and =W fun x -> fun y ->
  if x then if y then true else false else false;
```

```
11 or =W fun x -> fun y ->
12   if x then true else y;
13 not =W fun x -> if x then false else true;
14 xor =W fun x -> fun y -> (not (x = y));
15
16 (* 'a predecessor *)
17 pred =W fun x ->
18          if x = min : 'a then min : 'a else
19            ((iter y -> if (succ y) = x then inl(y) else
                inr(succ y)) min : 'a);
20
21 loop =U hack fun x ->
22   case x of
23     inl(y) ->
24       let y be <store, stepq> in
25         case stepq of
26           inl(argq) ->
27            let argq be <argstore, unit> in
28              inl(<store, inl(<argstore, store>)>)
29         | inr(continueOrStop) ->
30            case continueOrStop of
31              inl(continue) -> inl(<continue, inr(<>)>)
32            | inr(stop) -> inr(inr(stop))
33   | inr(z) ->
34     case z of
35       inl(basea) ->
36           let basea be <junk, basea> in
37             inl(<basea, inr(<>)>)
38     | inr(initialq) ->
39         inr(inl(<<>, <>>))
40  as {'a}({'c}['a] --o ['a + 'b]) --o ({'d}['a] --o ['b]);
41 continue =W fun x -> inl(x);
42 return =W fun x -> inr(x);
43 (* left means continue with the loop, right means stop with
     this value *)
44
45 fst =W fun x -> let x be <x1, x2> in x1;
46 snd =W fun x -> let x be <x1, x2> in x2;
```

```
47
48 ifU =U fun x -> fun y -> fun z->
49   let x be [c] in
50     case c of
51       inl(d) -> y
52     | inr(d) -> z;
53
54 (*========== SUBROUTINES ==========*)
55
56 (*Full adder: performs an addition operation on bit (two digits
       and a previous carry)
57 and produces a two-bit output (the sum plus the carry)*)
58 fa =W fun x -> fun y -> fun c ->
59    <xor x (xor y c),
60     or (and x y) (or (and y c) (and x c))>;
61
62 (*========== ALGORITHM ==========*)
63
64 (*When they ask me the i-th bit of the result, I recompute all
       the preceding colounms
65 to have the current carry. Then I add the two bits and the
       carry with the fa.
66 If I have reached the searched coloumn, return the fa result,
67 if not, continue using fa carry*)
68
69 addL =U (* Add: (['a] --o 2) --o (['a] --o [2]) --o (['a] --o
       2)*)
70  fun n1 : {'b1}['a] --o [1+1] ->
71  fun n2 : {'b2}['a] --o [1+1] ->
72   fun iU : ['a] ->
73     let iU be [i] in (*returned function, i will be ['a] and
          fun will return a boolean*)
74       loop (* for every iteration this loop calculates, for
             l=1,2,...i, the i-th bit of the result and the
             corresponding carry*)
75       (
76           fun v ->
77             let v be [vw] in
```

```
78                  let (n1 [fst vw]) be [x1] in
79                  let (n2 [fst vw]) be [x2] in
80                  (ifU ([(fst vw) = i])
81                  ([return(fst (fa  x1 x2 (snd vw)))]) (*it is
                        the searched digit, calculate and return it*)
82                    [continue(
83                      < (succ (fst vw)),
84                        (snd (fa x1 x2 (snd vw)))>
85                     )])
86       )
87       [< min : 'a , false>]; (*the first component is the l
             initial value (0), the second is the carry, also
             starting as 0 *)
88
89
90
91 (*========== TESTS ==========*)
92
93 w1 =U
94   fun x->
95      let x be [c] in
96               case c of
97               inl(u) -> [uno]
98              |inr(d_t) -> (
99                 case d_t of
100                inl(d) -> [uno]
101                |inr(t) -> [uno])
102               ;
103
104 w2 =U
105   fun x ->
106      let x be [c] in
107               case c of
108               inl(u) -> [uno]
109              |inr(d_t) -> (
110                 case d_t of
111                 inl(d) -> [uno]
112                |inr(t) -> [zero]);
```

```
113
114 testadd1 =U addL w1 w2 ([(min)]:[1+(1+1)]);
115 testadd2 =U addL w1 w2 ([(succ min)]:[1+(1+1)]);
116 testadd3 =U addL w1 w2 ([succ (succ min)]:[1+(1+1)]);
117
118 testadd1b =U addL w1 w1 ([(min)]:[1+(1+1)]);
119 testadd2b =U addL w1 w1 ([(succ min)]:[1+(1+1)]);
120 testadd3b =U addL w1 w1 ([succ (succ min)]:[1+(1+1)]);
```

3.4 Moltiplicazione

Vogliamo ora studiare la realizzazione di un algoritmo per la moltiplicazione di due parole binarie.

Ricordiamo l'algoritmo che ci è stato insegnato alla scuola elementare: si moltiplica il moltiplicando con ogni cifra del moltiplicatore, poi si sommano i risultati parziali opportunamente incolonnati.

Posta n la lunghezza dell'input, si vede subito che questo algoritmo *non può essere sublineare in spazio* rispetto a n: c'è infatti bisogno di memorizzare n risultati intermedi la cui lunghezza può essere al più $2n$.

3.4.1 L'Algoritmo

Senza scartare l'idea fornitaci dalle moltiplicazioni in colonna, cerchiamo di migliorare l'algoritmo evitando di memorizzare i risultati intermedi.

Supponiamo ora di sommare due parole binarie e che n sia il numero di bit di ciascuno dei due numeri in input. A questo punto cominciamo a sommare le colonne dei risultati parziali da sinistra a destra, tenendo sempre traccia del riporto. Per fare questo però non importa memorizzare le colonne: dimostriamolo. Se chiamiamo a_i e b_i l'i-esimo bit a partire da destra dei due operandi ($i = 0$ è dunque l'indice per il bit meno significativo) e se con c indichiamo il resto della somma della colonna precedente, allora è facile

convincersi che vale la seguente uguaglianza:

$$r_i = c + \sum_{j+k=i} a_j b_k$$

dove con c indichiamo il resto della colonna precedente.

Poiché stiamo lavorando su numeri in notazione binaria, il risultato della colonna sarà semplicemente il bit meno significativo di r_i (`r_i mod 2`) mentre il resto per la colonna successiva sarà composto da tutti gli altri bit di r_i (calcolabili equivalentemente con shift a destra, `div` o `floor`).

In un qualche pseudocodice con paradigma imperativo, possiamo scrivere:

```
// Array che contengono le rappresentazioni binarie
molt(a[0..n-1], b[0..n-1])
   x := 0
   for i from 0 to 2n-2
      for j from max(0,i+1-n) to min(i,n-1)
         k := i - j
         x := x + (a[j] * b[k])
      result[i] := x mod 2
      x := floor(x/2)
```

3.4.2 Spazio Logaritmico

Proviamo ora a dimostrare formalmente che questo nuovo algoritmo utilizza spazio logaritmico.

È facile vedere che gli indici j e k possono essere memorizzati usando $O(\log n)$ bit. Più complesso è stabilire se anche il valore di c e di r_i rimangono all'interno di questo bound spaziale.

Proposizione 3.4.1. *Il riporto non assume un valore maggiore di n e la somma non supera mai $2n$.*

Dimostrazione. Per induzione avremo:

- *caso base:* per la prima colonna ($i = 0$) il resto è zero

- *caso induttivo:* assumiamo *per ipotesi* che, per una generica colonna $(i = m)$, il resto sia al più n.
  Allora per la colonna successiva $(i = m + 1)$ avremo che, essendoci al massimo n bit in essa ed essendo, per ipotesi induttiva, il riporto al più n, la somma varrà al più $2n$, e il riporto di questa sarà al massimo (per come è definito l'algoritmo) la sua metà, cioè appunto n. Entrambi questi valori possono essere dunque memorizzati usando $O(\log n)$ bit.

□

3.4.3 Implementazione

La struttura del codice è simile a quella già descritta per l'addizione. La funzione `multL` avrà tipo

```
multL :U ([’a]-->[1+1])-->([’a]-->[1+1])-->[’a]--o[1+1]
```

e dunque restituirà una funzione che calcola, su richiesta, un bit del risultato.

Per fare ciò utilizza, come suggerito dall'algoritmo, due loop annidati. Il primo va da 0 a i, dove i è l'indice della cifra del risultato richiesta. Pensiamo all'algoritmo delle scuole elementari: il loop calcola per ogni colonna il risultato e il resto, e passa all'iterazione successiva quest'ultimo, oppure restituisce il risultato se abbiamo raggiunto la colonna desiderata.

Il calcolo delle somme parziali di ogni colonna viene realizzato con un ciclo annidato al primo, che ricalcola tali somme da 0 alla colonna corrente e le passa all'iterazione successiva, in una variabile di tipo `(1+1)*’a+1`. Infatti, come abbiamo già dimostrato, tali somme varranno al massimo $2n$, dove, in questo caso, $n = |\text{'a}|$.

3.4.4 Sorgente Completo

```
1 (*========== UTILS ==========*)
2
3 true =W inl(<>);
```

```
4  false =W inr(<>);
5
6  uno =W succ min :1+1;
7  zero =W min : 1+1;
8
9  and =W fun x -> fun y ->
10   if x then if y then true else false else false;
11 or =W fun x -> fun y ->
12   if x then true else y;
13
14 (* 'a predecessor *)
15 pred =W fun x ->
16          if x = min : 'a then min : 'a else
17            ((iter y -> if (succ y) = x then inl(y) else
                 inr(succ y)) min : 'a);
18
19 loop =U hack fun x ->
20   case x of
21     inl(y) ->
22       let y be <store, stepq> in
23         case stepq of
24           inl(argq) ->
25            let argq be <argstore, unit> in
26              inl(<store, inl(<argstore, store>)>)
27         | inr(continueOrStop) ->
28            case continueOrStop of
29              inl(continue) -> inl(<continue, inr(<>)>)
30            | inr(stop) -> inr(inr(stop))
31   | inr(z) ->
32     case z of
33       inl(basea) ->
34           let basea be <junk, basea> in
35             inl(<basea, inr(<>)>)
36     | inr(initialq) ->
37         inr(inl(<<>, <>>))
38  as {'a}({'c}['a] --o ['a + 'b]) --o ({'d}['a] --o ['b]);
39 continue =W fun x -> inl(x);
40 return =W fun x -> inr(x);
```

```
41 (* left means continue with the loop, right means stop with
       this value *)
42
43 fst =W fun x -> let x be <x1, x2> in x1;
44 snd =W fun x -> let x be <x1, x2> in x2;
45
46 ifU =U fun x -> fun y -> fun z->
47   let x be [c] in
48     case c of
49       inl(d) -> y
50     | inr(d) -> z;
51
52 (*========== SUBROUTINES ==========*)
53
54 (*Binary digit multiplication. *)
55 binmult =U fun x -> fun y ->
56   let x be [a] in
57   let y be [b] in
58     if (a = zero) then [zero] else [b];
59
60 (*Non negative subtraction between generic types 'a*)
61 subW =W fun x -> fun y ->
62  (iter z -> let z be <x1, y1> in
63                if or (y1 = min : 'a) (x1 = min : 'a ) then
                      inl(x1)
64                else inr(<pred x1, pred y1>))
65  <x,y>;
66
67 sub =U fun x -> fun y -> let x be [xc] in let y be [yc] in
       [subW xc yc];
68
69 (*add between two generic types 'a and 'b*)
70 (*
71 addW =W fun x -> fun y ->
72  (iter z -> let z be <x1, y1> in
73                if or (y1 = min : 'a) (succ x1 = x1) then inl(x1)
74                else inr(<succ x1, pred y1>))
75  <x,y>;
```

```
76
77 add =U fun x -> fun y -> let x be [xc] in let y be [yc] in
       [addW xc yc];
78 *)
79
80 (*Add 0 or 1 to a generic type 'a*)
81 addBin =U fun x -> fun y ->
82   let x be [a] in
83   let y be [b] in
84     if (b = uno) then [(succ a)] else [a];
85
86 (* Calculates integer division by 2 and its mod*)
87 divmod =U
88   fun x : ['a] ->
89     let x be [xw] in
90     loop
91     (
92       fun v ->
93        let v be [vw] in
94     (*(fst vw) will count iterations, i.e. the result of
          integer division. (snd vw) holds the partial results of
          subtraction and will become the mod *)
95         case (or ((snd vw) = min : 'a) ((snd vw) = (succ min :
              'a)) ) of
96              inl(true) -> [return(vw)]
97             |inr(false) -> [continue(<(succ (fst vw)), (pred
                  (pred (snd vw)))>)] (* < d++, m-2 >*)
98     )
99     ([< min : 'a, xw>] : ['a * 'a]); (*the first value will
          count iterations, i.e. the result of integer division.
          the second value holds the partial results of
          subtraction *)
100
101 floor =U fun x -> let (divmod x) be [c] in [fst c];
102 mod2  =U fun x -> let (divmod x) be [c] in [if ((snd c) = min :
      'a) then zero else uno ];
103
104
```

```
105 (*========== ALGORITHM ==========*)
106
107 (*
108 molt(a[0..n-1], b[0..n-1]) // Array che contengono le
        rappresentazioni binarie
109     x := 0
110     for i from 0 to 2n-2
111         for j from max(0,i+1-n) to min(i,n-1)
112             k := i - j
113             x := x + (a[j] * b[k])
114         result[i] := x mod 2
115         x := floor(x/2)
116 *)
117
118 get_l =W fun x -> (fst x);
119 get_c =W fun x -> (snd x);
120
121 multL =U (* Multiply: (['a] --o 2) --o (['a] --o [2]) --o (['a]
        --o 2)*)
122  fun n1 : {'b1}['a] --o [1+1] ->
123  fun n2 : {'b2}['a] --o [1+1] ->
124   fun iU : ['a] ->
125     let iU be [i] in (*returned function, i will be ['a] and
            fun will return a boolean*)
126       loop (* for every iteration this loop calculates, for
            l=1,2,...i, the i-th bit of the result and the
            corresponding carry*)
127       (
128           fun v  (*: [('a * (1+1)) * ((1+1)*'a+1)]*) ->
129             let v be [vw] in (* we have <l,c> where l is the
                iterator and c is the carry*)
130               copy (
131                   loop (* this loop calculates, for j from 0 to
                        l, the sum of products on a coloumn *)
132                   (
133                         fun d ->
134                          let d be [dw] in  (*j is the loop
                              iterator, ctmp is the partial sum
```

```
                                of a coloumn*)
135                             let (addBin
136                                     [(snd dw)]
137                                     (binmult (n1 [(fst dw)])
                                            (n2 (sub [(get_l vw)]
                                            [(fst dw)])))  )
138                             be [c1w] in
139                             (*to simplify, the loop goes from 0
                                    to l*)
140                             case ((get_l vw) = (fst dw)) of
141                               inl(true) -> [return(c1w)]
142                              |inr(false) ->
143                              [continue(
144                                 < succ (fst dw),
145                                   c1w >)]
146                         )
147                         [< min : 'a, (get_c vw) >](* first
                                component is loop iteraror j, the second
                                partial sum of the i-th column, up to
                                row j (and starts from the previous
                                carry)*)
148                 )
149                 as o1,o2 in
150                 let (mod2 o1) be [result] in
151                 let (floor o2) be [carry] in
152                 (ifU ([(get_l vw) = i])
153                 ([return(result)])
154                 (
155                 [continue(
156                   <(succ (get_l vw)), (*l++*)
157                     carry>    (*carry*)
158                 )]))
159         )
160         ([< (min : 'a), min : (1+1)*'a+1>] ); (*the first
                component is the l initial value (0), the second is
                the carry, also starting as 0 *)
161
162 (*========== TESTS ==========*)
```

```
163
164 w1w =W fun x ->
165   if (x = min : 1+1+1+1+1+1+1+1 ) then uno
166   else if (x = (succ min : 1+1+1+1+1+1+1+1)) then uno
167   else if (x = (succ (succ min : 1+1+1+1+1+1+1+1))) then uno
168   else zero;
169
170
171 w2w =W fun x ->
172   if (x = min : 1+1+1+1+1+1+1+1) then uno
173   else if (x = (succ min : 1+1+1+1+1+1+1+1)) then uno
174   else if (x = (succ (succ min : 1+1+1+1+1+1+1+1))) then zero
175   else zero;
176
177 w1 =U fun x -> let x be [c] in [w1w c];
178 w2 =U fun x -> let x be [c] in [w2w c];
179
180 tm1 =U multL w1 w2 ([min : 1+1+1+1+1+1+1+1]);
181 tm2 =U multL w1 w2 ([succ min : 1+1+1+1+1+1+1+1]);
182 tm3 =U multL w1 w2 ([(succ (succ min : 1+1+1+1+1+1+1+1))]);
```

Conclusioni e Sviluppi Futuri

"I can only tell you about the future
and let you draw your own conclusions."
– Janet Reno

Al termine di questa tesi, è bene riepilogare la strada percorsa, gli ostacoli incontrati lungo il cammino e gli orizzonti che si aprono da questo punto in poi.

Cosa c'era e Cosa c'è Ora

Siamo partiti dall'analisi dei domini in cui è necessario o conveniente porre dei limiti, in termini di spazio utilizzato, alla computazione di un determinato programma software. Ci siamo presto resi conto che, in un mondo dominato dalla Rete Internet e dal Web, è naturale avere a che fare con collezioni di dati *enormi* e che magari risiedono su server remoti. Immediata conseguenza è quindi la necessità di poter scrivere programmi che operino su questi dati, pur non potendo questi dati entrare *tutti insieme* nella memoria di lavoro del programma stesso.

Abbiamo a questo punto fatto un passo indietro, per studiare le nozioni di Informatica Teorica, e in particolare di Teoria della Complessità, che stanno dietro ai limiti spaziali di un problema. Abbiamo per questo utilizzato un modello alternativo a quello della Macchina di Turing, che di solito si usa in

questo ambito, chiamato Macchina di Turing Offline. Pur avendo dimostrato che questo modello ha lo stesso potere espressivo della MdT, esso ci permette di analizzare algoritmi che utilizzano spazio sub-lineare rispetto al loro input. In questo modello infatti l'input e l'output non sono memorizzati, ma richiesti e restituiti da e verso l'ambiente in maniera *interattiva*.

Questo strumento ci ha permesso di passare a un nuovo "stile" di scrittura del software: la *programmazione bidirezionale*. È subito chiaro però che per programmare con questo stile bisogna modificare abbastanza radicalmente il modo di pensare agli algoritmi, oltre a dover prestare molta attenzione alle interazioni funzione-funzione e funzione-ambiente.

Un linguaggio di programmazione di recentissima introduzione, IntML, si propone di superare questi limiti fornendo al programmatore delle primitive per programmare in modo "naturale", demandando poi al linguaggio stesso la *traduzione* da stile unidirezionale a stile bidirezionale. Viene inoltre dimostrato che grazie a questo modello, IntML permette di scrivere programmi che utilizzano uno spazio logaritmico di memoria rispetto all'input.

Questo linguaggio è definito formalmente e in modo teorico negli articoli di ricerca che lo introducono [DLS10a, DLS10b], ma contemporaneamente possiede un'implementazione concreta che può essere utilizzata per scrivere effettivamente programmi *logspace*. Essendo l'interprete ancora in una forma prototipale, non esisteva alcuna guida alla sintassi concreta dei costrutti messi a disposizione. È stata cura di questa tesi dunque realizzare un "manuale" che si focalizzasse sopratutto sugli aspetti di sintassi e semantica concreta del linguaggio, e che fornisse alcuni esempi di utilizzo di IntML, tralasciando magari i dettagli implementativi e formali, che, pur essendo fondamentali, rischiano di confondere il programmatore abituato ad usare linguaggi tradizionali.

Infine, per mostrare come IntML permetta davvero di scrivere facilmente funzioni e programmi *logspace*, abbiamo realizzato una *libreria di funzioni aritmetiche* che operano in spazio logaritmico. Queste mettono a disposizione del programmatore alcune routine basilari, ma non presenti nel linguaggio,

che possono poi essere utilizzate "così come sono" per scrivere altri programmi, avendo la garanzia del loro uso limitato di spazio. In particolare, abbiamo scritto funzioni per calcolare divisione intera e resto sui naturali, e funzioni per confrontare, sommare e moltiplicare numeri espressi come parole binarie.

Difficoltà Incontrate

Nell'affrontare questo lavoro sono state incontrate alcune difficoltà.

Ci siamo subito accorti che i testi di Algoritmi utilizzati nei corsi universitari soffermano la loro attenzione soprattutto sui problemi legati alla complessità in termini di tempo, cercando di fornire algoritmi efficienti da questo punto di vista, senza preoccuparsi di ottimizzare - e spesso nemmeno di analizzare - quanto questi algoritmi siano o meno efficienti dal punto di vista dello spazio di calcolo utilizzato. Questo è dovuto all'altrettanto sorprendente fatto che, ancora oggi, per molti problemi anche relativamente semplici non esistono studi concreti sulla possibilità di risolverli in spazio limitato, magari sublineare.

Per quanto riguarda la programmazione in IntML, le principali difficoltà incontrate sono dovute all'utilizzo del paradigma funzionale puro, che si contrappone al modo naturale ("imperativo") di pensare agli algoritmi, e richiede invece un approccio più matematico. Altre difficoltà incontrate sono dovute ai limiti, dettati dalla necessità di garantire l'utilizzo logaritmico di spazio, che IntML pone ad esempio al passaggio e all'uso di parametri.

È altresì vero che, con una discreta conoscenza del paradigma funzionale e dell'idea che sta alla base della computazione bidirezionale, è stato abbastanza semplice implementare la libreria di funzioni in IntML.

Sviluppi Futuri

Come già visto nell'introduzione, sono molti gli ambiti *potenziali* in cui un lavoro di questo tipo può essere utilizzato. È ovvio che, essendo IntML

un linguaggio di recentissima introduzione, moltissimi aspetti possono essere sviluppati e potenziati.

Dal punto di vista degli studi teorici, è sicuramente necessario concentrare l'attenzione sulla complessità in termini di spazio, di modo da fornire al programmatore algoritmi sublineari da poter implementare e utilizzare nei propri lavori.

Per quanto riguarda il linguaggio, esso parte da una base molto semplice. Si potrebbe sicuramente sostituire la Working Class con un linguaggio più espressivo, e d'altro canto aumentare il numero di costrutti che l'Upper Class mette a disposizione, per facilitare e rendere ancora più naturale la programmazione in IntML.

Molte features possono essere aggiunte all'interprete, sia per migliorarne l'usabilità (interfaccia grafica, evidenziazione della sintassi...) che per facilitare il compito del programmatore (messaggi di errori chiari ed esplicativi, debugger...).

La semplicità di IntML, la cui curva di apprendimento sale rapidamente, lo fanno anche un ottimo candidato per l'uso didattico, magari in corsi avanzati in cui si trattano aspetti meno conosciuti della complessità computazionale. Per incentivarne e facilitare l'uso didattico potrebbero essere inoltre realizzati tool che rappresentano graficamente il flusso della computazione, per meglio comprenderne e analizzarne il funzionamento.

Nell'immediato, ci si sta proponendo di estendere la libreria ora disponibile con numerose funzioni che semplificano il lavoro di programmazione (pensiamo ad esempio agli algoritmi di ordinamento, all'utilizzo di strutture dati quali grafi, ecc.).

Di certo la diffusione e quindi anche il conseguente sviluppo e miglioramento del linguaggio possono essere trainati in modo estremamente positivo dalla scelta di IntML per qualche applicazione concreta (didattica o tecnologica), in modo da renderlo magari più visibile al "grande pubblico".

Bibliografia

[AB09] S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[CM98] G. Cousineau and M. Mauny. *The functional approach to programming*. Cambridge Univ. Press, 1998.

[DG06] A. Dovier and R. Giacobazzi. Dispense per il corso di fondamenti dell'informatica: Linguaggi formali, calcolabilità e complessità, 2006. URL: `www.dimi.uniud.it/~dovier/DID/dispensa.pdf`.

[DLS10a] U. Dal Lago and U. Schöpp. Functional programming in sublinear space. *European Symposium on Programming (ESOP2010), 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), Shangai, China*, 2010.

[FFFK03] M. Felleisen, M. Findler, M. Flatt, and S. Krishnamurthi. *How To Design Programs, an introduction to programming and computing*. The MIT Press, 2003.

[MG06] S. Martini and M. Gabbrielli. *Linguaggi di programmazione: principi e paradigmi*. McGraw-Hill Italia, 2006.

[San10] Rahul Santhanam. Courses notes of computational complexity, lecture 2, 2010. URL: http://www.inf.ed.ac.uk/teaching/courses/cmc/.

[Sip05] Michael Sipser. *Introduction to the Theory of Computation*. Course Technology, 2005.

Ringraziamenti

"I can no other answer make, but, thanks, and thanks."
– William Shakespeare

Al Dottor Ugo Dal Lago, che mi ha insegnato, aiutato, corretto e spronato più di quanto potessi aspettarmi;
al Dottor Ulrich Schöpp, il cui aiuto in alcuni momenti è stato davvero *provvidenziale*;
ai miei insostituibili Mamma e Papà, che mi hanno insegnato ad essere la persona che sono, pur lasciandomi essere come volevo;
alla mia sorellona Milena, che mi ha fatto conoscere l'Informatica, e ai miei bellissimi nipotini Thomas e Francesco, che sono già sulla buona strada per impararla meglio di me;
a tutti quegli insegnanti delle scuole Elementari, Medie, Superiori e dell'Università che hanno creduto in me e mi hanno trasmesso la passione per il Sapere;
ai miei amici di sempre: Laura, Giacomo, Andrea, senza i quali la mia vita non avrebbe senso; ad Alessandro e Deborah, che mi hanno regalato preziosi momenti di svago durante questi mesi; a Morelli, Battista e Manto, che hanno saputo accettare le mie assenze e con cui mi sono divertito un mondo quando ero presente;
a tutti i colleghi e amici dell'Università, e in particolare a Federica, Francesco e Federico, senza i quali questo viaggio sarebbe di certo più difficile e noioso;
a tutti voi va il mio infinito *grazie*.