

ALMA MATER STUDIORUM · UNIVERSITÀ DI  
BOLOGNA

---

SCUOLA DI SCIENZE  
Corso di Laurea in Informatica

# Da Matita a Dedukti e ritorno

Relatore:  
Chiar.mo Prof.  
Claudio Sacerdoti Coen

Presentata da:  
Mattia Girolimetto

I Appello di Laurea  
Anno Accademico 2022-2023



*A Clarissa, per avermi cambiato*

*A Cristina e Teodoro,  
per avermi dato questa opportunità*



*La logica, come il whisky, perde  
il suo effetto benefico se assunta  
in quantità eccessive.*

---

*Lord Dunsany*



## SOMMARIO

Dedukti è un software che agisce da type checker per definizioni, enunciati e prove codificati nel calcolo che implementa. Quest'ultimo è sufficientemente espressivo da poter rappresentare anche altri calcoli. Questo rende Dedukti un framework logico con il quale è possibile sviluppare un sistema di codifica che converta nel suo linguaggio le dimostrazioni scritte per altri proof assistant. Inizialmente si effettua un'analisi del processo di esportazione del codice da Matita a Dedukti, Matita è un assistente alla dimostrazione sviluppato presso l'Università di Bologna, basato sul calcolo delle costruzioni (co)-induttive. Successivamente si illustra la funzionalità di importazione del codice Dedukti in Matita e si esaminano le sfide che si presentano nel ricostruire i termini originali di Matita, a partire dal codice Dedukti esportato in precedenza.





# INDICE

1	INTRODUZIONE	11
2	PREREQUISITI	13
2.1	La teoria dei tipi . . . . .	13
2.1.1	La teoria dei tipi . . . . .	13
2.1.2	L'isomorfismo di Curry-Howard . . . . .	14
2.1.3	La teoria dei tipi nella pratica . . . . .	14
2.2	Dedukti e Matita . . . . .	15
2.2.1	Dimostratori interattivi di teoremi . . . . .	15
2.2.2	Dedukti . . . . .	16
2.2.3	Matita . . . . .	18
3	DA MATITA A DEDUKTI	21
3.1	Krajono . . . . .	21
3.1.1	Integrazione in Matita . . . . .	21
3.2	L'export . . . . .	22
3.2.1	Il processo di esportazione . . . . .	22
3.2.2	L'encoding . . . . .	22
3.3	I problemi dell'export . . . . .	24
3.3.1	I problemi di Krajono . . . . .	24
3.3.2	Perdita di informazione . . . . .	24
4	DA DEDUKTI A MATITA	27
4.1	Dedukti in Matita . . . . .	27
4.1.1	Il parser Dedukti . . . . .	27
4.2	L'import . . . . .	28
4.2.1	La gestione dei nomi . . . . .	28

## Indice

4.2.2	Definizioni e Dichiarazioni . . . . .	28
4.3	Invertire l'esportazione . . . . .	29
4.3.1	L'uso delle pragma . . . . .	30
4.3.2	Costanti CIC . . . . .	31
4.3.3	Punto fisso . . . . .	31
4.3.4	Tipi induttivi . . . . .	33
4.3.5	Codice generato . . . . .	34
5	CONCLUSIONI . . . . .	35
5.1	Conclusioni . . . . .	35
5.2	Sviluppi futuri . . . . .	35
	BIBLIOGRAFIA . . . . .	37

# 1 INTRODUZIONE

I *proof assistant* sono software in grado di verificare una dimostrazione formale scritta da un utente. Questa viene rappresentata internamente attraverso termini di un linguaggio funzionale dotato di un sistema di tipi avanzato, tale da renderlo equivalente a una logica. Il loro sviluppo, negli anni, ha portato a una frammentazione della conoscenza in quanto sistemi diversi, che usano logiche diverse, non sono compatibili tra loro. Per ovviare a questo problema un gruppo di ricercatori dell'INRIA ha creato *Dedukti*, la cui idea è quella di agire da *type checker* per definizioni, enunciati e prove codificati nel calcolo che implementa. Tale calcolo è sufficientemente espressivo per rappresentare in *Dedukti* qualunque sistema di tipi, ovvero qualunque linguaggio funzionale con un sistema di tipi avanzato. Questo rende *Dedukti* un *logical framework*, ovvero un calcolo che svolge il ruolo di meta-calcolo per la rappresentazione uniforme di altri calcoli. Grazie a esso è possibile sviluppare un sistema di codifica che converta dimostrazioni scritte nel linguaggio di altri proof assistant, come *Coq* e *HOL lite*, nel metalinguaggio di *Dedukti*. Proprio per questo all'INRIA è stato sviluppato *Krajono*, un *fork* di *Matita*, che a sua volta è un proof assistant in sviluppo nel Dipartimento di Informatica Scienze e Ingegneria dell'Università di Bologna. *Krajono* permette di esportare le definizioni e i teoremi dati in *Matita* verso *Dedukti*, attraverso una codifica del calcolo di *Matita* in quello di *Dedukti*. *Krajono* tuttavia non è più mantenuto e pertanto non integra le funzionalità del *Matita baseline* sviluppate negli ultimi anni. Il lavoro di questa tesi è diviso in due parti: la prima prevede l'integrazione della funzionalità di esportazione di *Krajono* nel *Matita* odierno, mentre l'obiettivo della seconda consiste nell'implementare la possibilità di reimportare codice *Dedukti* esportato precedentemente da *Matita*. Il secondo punto in particolare ha richiesto la risoluzione di una serie di problemi, in primis dovuti al

## *1 Introduzione*

fatto che il linguaggio di Matita sia più rigido e strutturato del metalinguaggio di Dedukti, pertanto durante l'esportazione vengono perse informazioni necessarie alla ricostruzione del codice originale. Ciò è stato risolto dotando l'export di un meccanismo per preservarle dove necessario. In secondo luogo i sistemi di tipi codificati in Dedukti, per essere consistenti, richiedono che le regole di riscrittura scelte dagli sviluppatori per implementare la codifica siano confluenti e normalizzanti. Dedukti non effettua questa verifica, che demanda a altri strumenti. Matita, invece, impone un insieme molto restrittivo di controlli sintattici per garantire confluenza e terminazione, che sono in generale proprietà indecidibili. Pertanto, quando si inverte il processo di esportazione, sorge la necessità di ricostruire termini che soddisfino le condizioni sintattiche di Matita, che sono sufficienti, ma non necessarie, e che potrebbero essere state violate durante la codifica in Dedukti, nonostante le regole fossero anche confluenti e normalizzanti.

# 2 PREREQUISITI

In questo capitolo vengono trattati prerequisiti necessari per la comprensione del lavoro esposto nei capitoli quelli successivi. Nella sezione 2.1 sono illustrati i fondamenti della *Teoria dei Tipi*, teoria alla base di questa tesi. Nella sezione 2.2 vengono presentati Matita e Dedukti, i software protagonisti di questo lavoro.

## 2.1 LA TEORIA DEI TIPI

### 2.1.1 LA TEORIA DEI TIPI

La *teoria dei tipi* è una branca della logica e dell'informatica teorica il cui obiettivo è quello di studiare i cosiddetti *type system*, ovvero insiemi di regole che associano una proprietà chiamata *tipo* a degli oggetti chiamati *termini*. Intuitivamente, assegnare un tipo a un termine significa assegnare al termine un'etichetta che rappresenta la natura del termine stesso. Esempi comuni possono essere:

- 42 è un numero naturale
- $-5$  è un numero intero
- *false* è un valore di verità

Formalmente si usa rappresentare queste espressioni separando il termine dal tipo usando il simbolo “:”.

In teoria dei tipi, anche le funzioni sono termini e possono essere a loro volta tipizzate. Ad esempio, la seguente funzione rappresentata attraverso un  $\lambda$ -termine del  $\lambda$ -calcolo di Church

## 2 Prerequisiti

$$(\lambda x : \mathbb{N} . x + x)$$

è definita dall'insieme  $\mathbb{N}$  a  $\mathbb{N}$  stesso, e pertanto si dice che ha tipo  $\mathbb{N} \rightarrow \mathbb{N}$

### 2.1.2 L'ISOMORFISMO DI CURRY-HOWARD

Durante il '900 i logici Haskell Curry e William Alvin Howard, scoprirono una corrispondenza diretta tra prove formali e programmi. In particolare notarono che gli operatori logici e le regole usate durante una dimostrazione formale sono equivalenti a tipi e costrutti usati nei programmi scritti usando linguaggi di programmazione funzionali. Ne segue che il verificare la correttezza di una prova corrisponde al verificare la correttezza degli assegnamenti di tipo di un programma. Nella sua formulazione più generale, l'isomorfismo di Curry-Howard può essere riassunto con la seguente tabella:

<b>Operatori logici</b>	<b>Tipi</b>
$\top$	Tipo unit
$\perp$	Tipo vuoto/void
$\wedge$	Tipi prodotto
$\vee$	Tipi somma
$\Rightarrow$	Tipi funzione
$\exists$	Tipi $\Sigma$
$\forall$	Tipi $\Pi$

### 2.1.3 LA TEORIA DEI TIPI NELLA PRATICA

La teoria dei tipi trova quindi grande applicazione nel campo dell'informatica grazie allo studio e allo sviluppo dei linguaggi di programmazione. Inoltre, ha permesso lo sviluppo di dimostratori interattivi di teoremi grazie all'isomorfismo di Curry-Howard.

## 2.2 DEDUKTI E MATITA

### 2.2.1 DIMOSTRATORI INTERATTIVI DI TEOREMI

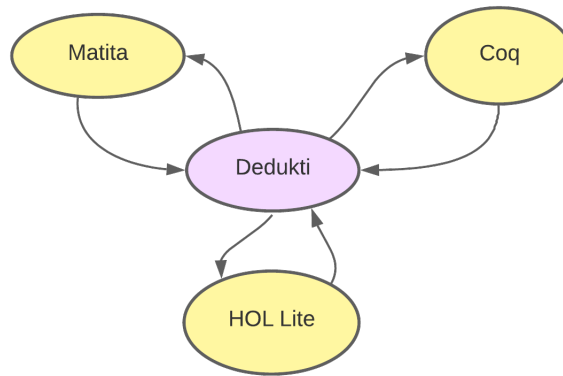
Un dimostratore interattivo di teoremi (o *proof assistant*) è un software che permette all'utente di costruire e verificare delle dimostrazioni matematiche formali. Presa in input una prova espressa utilizzando uno specifico linguaggio formale il software è in grado di verificarne la correttezza. In questo modo si possono costruire dimostrazioni in modo interattivo, controllando progressivamente la correttezza di ogni passo. Uno dei benefici chiave dell'usare un dimostratore interattivo automatico è l'abilità di eliminare gli errori e le ambiguità che possono comparire nelle dimostrazioni tradizionali.

Durante la fase di verifica, il proof assistant rappresenta i termini della dimostrazione utilizzando un linguaggio di programmazione dotato di un sistema di tipi avanzato, ottenendo alla fine una rappresentazione che può essere considerata equivalente a un programma. Successivamente esegue un processo di *type checking* in cui verifica la correttezza degli assegnamenti di tipo del programma. Grazie all'isomorfismo di Curry-Howard questo equivale a verificare la correttezza della prova.

Il numero di proof assistant è aumentato nel tempo. Ciò è sicuramente un segnale positivo per la comunità scientifica, in quanto dimostra un crescente interesse verso lo sviluppo di questi strumenti. Tuttavia, questo aumento, unito al fatto che proof assistant diversi sfruttano logiche diverse, porta a una frammentazione della conoscenza. Per un utente non è quasi mai possibile, infatti, dimostrare la veridicità di un teorema usando un proof assistant e usare la stessa dimostrazione in un altro di questi tool senza doverla riscrivere. Questo perché diversi proof assistant usano diversi sistemi di tipi e una dimostrazione, per poter essere verificata con sistema di tipi diverso, deve essere tradotta. Nasce dunque l'esigenza di favorire l'interoperabilità tra questi sistemi, in modo da arginare questo problema e favorire lo sviluppo scientifico.

### 2.2.2 DEDUKTI

Dedukti<sup>1</sup> è un *logical framework*, ovvero un meta-sistema che consente di definire logiche e teoremi usando tali logiche. È sviluppato da alcuni ricercatori del *Institut National de Recherche en Informatique et en Automatique* francese. Il software è open source, scritto nel linguaggio di programmazione OCaml e distribuito secondo i termini della CeCILL-B License. Uno degli obiettivi principali di Dedukti è creare una connessione tra i diversi sistemi di dimostrazione assistita. Ciò significa che le dimostrazioni possono essere tradotte da un sistema all'altro, agevolandone lo scambio e il riutilizzo tra ambienti di lavoro diversi. Infatti alcuni proof assistant, come ad esempio Coq e HOL Lite<sup>2</sup>, godono già della possibilità di esportare e importare codice da e verso Dedukti.



$\lambda\Pi$ -CALCOLO MODULO Alla base di questo logical framework c'è il  $\lambda\Pi$ -calcolo modulo (o semplicemente  $\lambda\Pi$  modulo), un'estensione del  $\lambda$ -calcolo che introduce la *tipizzazione dipendente* e le *regole di riscrittura*. La prima consente la specifica di tipi complessi che dipendono dai valori delle espressioni, mentre le seconde consentono la trasformazione di un'espressione in un'altra, seguendo determinate sostituzioni o manipolazioni. La sintassi del  $\lambda\Pi$  modulo è la seguente

---

<sup>1</sup>“Dedurre” in Esperanto

<sup>2</sup>Coq: <https://github.com/Deducteam/CoqInE>,  
HOL Lite: <https://arxiv.org/pdf/1507.08720.pdf>



*Termini*  $A, B, t, u ::= \text{Kind} \mid \text{Type} \mid \Pi x : A . B \mid \lambda x : A . B \mid A B \mid x$   
*Contesto*  $\Gamma ::= \emptyset \mid \Gamma, x : A \mid \Gamma, t \hookrightarrow u$

Mentre il sistema di tipi è definito dalle seguenti regole:

$$\frac{}{[\ ] \text{ ben formato}} \text{Contesto vuoto}$$

$$\frac{\Gamma \vdash A : \text{Type}}{\Gamma, x : A \text{ ben formato}} \text{Dichiarazione (Type)} \quad x \notin \Gamma$$

$$\frac{\Gamma \vdash A : \text{Kind}}{\Gamma, x : A \text{ ben formato}} \text{Dichiarazione (Kind)} \quad x \notin \Gamma$$

$$\frac{\Gamma \vdash t \hookrightarrow u \text{ ben formato}}{\Gamma, t \hookrightarrow u \text{ ben formato}} \text{Regola}$$

$$\frac{\Gamma \text{ ben formato}}{\Gamma \vdash \text{Type} : \text{Kind}} \text{Sorta}$$

$$\frac{\Gamma \text{ ben formato} \quad x : A \in \Gamma}{\Gamma \vdash x : A} \text{Variabile}$$

$$\frac{\Gamma \vdash A : \text{Type} \quad \Gamma, x : A \vdash B : \text{Type}}{\Gamma \vdash \Pi x : A . B : \text{Type}} \text{Prodotto (Type)}$$

$$\frac{\Gamma \vdash A : \text{Type} \quad \Gamma, x : A \vdash B : \text{Kind}}{\Gamma \vdash \Pi x : A . B : \text{Kind}} \text{Prodotto (Kind)}$$

$$\frac{\Gamma \vdash A : \text{Type} \quad \Gamma, x : A \vdash B : \text{Type} \quad \Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A . t : \Pi x : A . B} \lambda\text{-astrazione}$$

$$\frac{\Gamma \vdash t : \Pi x : A . B \quad \Gamma \vdash u : A}{\Gamma \vdash (t u) : B [u/x]} \text{Applicazione}$$

$$\frac{\Gamma \vdash A : s \quad \Gamma \vdash x : A \quad A \equiv_{\beta\Gamma} B}{\Gamma \vdash x : B} \text{Equivalenza}$$

## 2 Prerequisiti

**REGOLE DI RISCrittURA** Le regole di riscrittura definiscono come trasformare o riscrivere espressioni o termini secondo determinate regole specificate. Sono utili, ad esempio, per implementare il processo di  $\beta$ -riduzione, che rappresenta il passo di applicazione di una funzione nei calcoli lambda:

$$(\lambda x : A . f x) y \hookrightarrow f y$$

Tuttavia, l'introduzione delle regole di riscrittura nel calcolo  $\lambda$ II lo rende non *consistente*. Questo perché non vengono garantite le proprietà di *confluenza* e *normalizzazione*.

La proprietà di confluenza (o di Church-Rosser) afferma che se ci sono due riduzioni distinte o sequenze di riduzioni che possono essere applicate allo stesso termine, allora esiste un termine che è raggiungibile da entrambi i risultati, applicando sequenze (possibilmente vuote) di riduzioni aggiuntive. In altri termini, se esiste  $a$  tale che  $a \hookrightarrow^* b$  e  $a \hookrightarrow^* c$  allora esiste  $d$  tale che  $b \hookrightarrow^* d$  e  $c \hookrightarrow^* d$ .

La proprietà di normalizzazione invece afferma che ogni termine può essere ridotto a una forma normale in un numero finito di passi di riduzione. In altre parole, non ci sono sequenze infinite di riduzioni per un termine specifico. Questa proprietà è fondamentale per garantire la decidibilità di un calcolo.

### 2.2.3 MATITA

Matita è un proof assistant in sviluppo nel dipartimento di Informatica Scienze e Ingegneria dell'Università di Bologna. È open source, scritto nel linguaggio di programmazione OCaml ed è rilasciato secondo i termini della GNU General Public Licence. È basato sul *calcolo delle costruzioni (co)induttive* (CIC), una teoria di tipi dipendenti che estende il *calcolo delle costruzioni* sviluppato da Thierry Coquand aggiungendo i tipi induttivi, ovvero tipi autoreferenziali. Le regole di tipaggio del CIC per i tipi semplici sono quelle comuni dei linguaggi di programmazione. Di seguito sono riportate quelle per le  $\lambda$ -astrazioni e le applicazioni

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash (\lambda x : A . M) : A \rightarrow B} \text{ } \lambda\text{-astrazione}$$

$$\frac{\Gamma \vdash f : A \rightarrow B \quad \Gamma \vdash x : A}{\Gamma \vdash (f x) : B} \text{ Applicazione}$$

CIC è più ricco del  $\lambda\Pi$  modulo usato da Dedukti in quanto in più consente di definire *tipi induttivi*, *punti fissi* e *pattern matching*.

I tipi induttivi consentono la creazione di tipi complessi e autoreferenziali e possono essere utilizzati per definire strutture matematiche illimitate come liste, alberi o grafi. L'esempio seguente mostra come si possa definire una lista concatenata di numeri naturali usando Matita. Una struttura di questo tipo può essere vista come una lista vuota oppure una coppia formata da un numero naturale e un'altra lista concatenata:

```
include "basics/pts.ma".

inductive list : Type[0]  $\stackrel{\text{def}}{=}$ 
  E : list
  | C : nat  $\rightarrow$  list  $\rightarrow$  list.
```

I punti fissi sono utili per definire funzioni ricorsive e, con l'aggiunta del pattern matching, permettono di definire funzioni ricorsive strutturali. Un esempio concreto di ciò è l'implementazione di una funzione per calcolare la lunghezza di una lista basata sul pattern matching per ragionare sulla struttura della lista stessa:

```
let rec len l on l  $\stackrel{\text{def}}{=}$ 
  match l with
  [ E  $\Rightarrow$  0
  | C hd t1  $\Rightarrow$  1 + len t1
  ].
```



# 3 DA MATITA A DEDUKTI

In questo capitolo viene prima introdotto *Krajono*, il fork di Matita da cui proviene il codice responsabile dell'export da Matita verso Dedukti. Successivamente viene analizzato nel dettaglio tecnico e teorico, il funzionamento di questa esportazione. Infine ne vengono discusse problematiche e limitazioni.

## 3.1 KRAJONO

Un team di ricercatori del *Institut national de recherche en informatique et en automatique* ha sviluppato Krajono<sup>1 2</sup>. Krajono ha la possibilità di esportare definizioni e teoremi dati in Matita verso Dedukti. Per far ciò sfrutta una codifica del calcolo di Matita nel metalinguaggio di Dedukti. Tuttavia questo fork non è più mantenuto da anni.

### 3.1.1 INTEGRAZIONE IN MATITA

In prima battuta, questo progetto consiste nell'integrare le funzionalità di esportazione di Krajono nella versione base di Matita. Tali funzionalità sono state ottenute copiando i singoli sorgenti responsabili dell'esportazione all'interno di Matita.

Integrare direttamente queste funzionalità tramite *pull request* tra i *repository Git*, comporterebbe complicazioni. Questo è principalmente dovuto al fatto che Krajono non è stato mantenuto da anni ed è basato su un fork separato di Matita chiamato *Matita with embedded Elpi*<sup>3</sup>.

---

<sup>1</sup>Significa "Matita" in Esperanto

<sup>2</sup><https://github.com/Deducteam/Krajono>

<sup>3</sup><https://github.com/LPCIC/matita>

## 3.2 L'EXPORT

### 3.2.1 IL PROCESSO DI ESPORTAZIONE

*Matitac* è il compilatore da linea di comando di Matita, se lo si lancia compila tutti i file con estensione *.ma* presenti nella directory corrente. Oppure è possibile compilare un singolo file passando come argomento il suo nome.

Krajono fornisce la possibilità di attivare la funzionalità di export specificando l'argomento `-extract_dedukti` a *Matitac*, sia lavorando con un unico file sia con un'intera directory. A processo concluso si potranno trovare, nella directory dei sorgenti, i file *.dk* contenenti il codice matita esportato.

Quando il flag menzionato è attivo, il motore di Matita avvia il processo di esportazione chiamando una specifica funzione da uno dei moduli OCaml importati da Krajono. Questi moduli compiono delle analisi del tipo e della struttura del termine Matita, con lo scopo di capire come applicare l'encoding appropriato. Una volta costruiti, i termini Dedukti risultanti dalla codifica vengono inseriti in una tabella hash, che alla fine del processo verrà scritta sul file *.dk* di output.

È importante notare come il processo di esportazione non è sempre un encoding *uno a uno*, ma determinate tipologie di termini Matita vengono codificati in più termini Dedukti.

### 3.2.2 L'ENCODING

La codifica è composta da una serie di definizioni ed è contenuta dentro un modulo Dedukti chiamato "cic". Comprendere il funzionamento di questo modulo è essenziale per eseguire l'operazione di importazione descritta nella sezione 4.3, quindi verranno esaminati i punti più importanti.

Il modulo inizia fornendo le definizioni necessarie per rappresentare i numeri naturali e l'operazione di massimo tra due numeri:

```
Nat : Type .  
  
z : Nat .  
s : Nat -> Nat .
```

```
def m : Nat -> Nat -> Nat.
[i] m i z --> i.
[j] m z j --> j.
[i, j] m (s i) (s j) --> s (m i j).
```

È interessante vedere all'opera le regole di riscrittura per definire la costante `m`.

In seguito si procede alla codifica degli universi. Innanzitutto viene definito il tipo `Sort` per rappresentare le sorte e successivamente vengono definite la sorta `prop` e una funzione per rappresentare i diversi universi.

```
Sort : Type.

prop : Sort.
type : Nat -> Sort.
```

Il naturale che `type` richiede in input rappresenta il numero dell'universo. Ad esempio, l'universo `Type[3]` viene codificato come

```
type (s (s (s z)))
```

Più avanti nel modulo si passa alla codifica dei tipi. Viene fornita la definizione di `Univ`, che accetta una sorta e restituisce un `Type` che rappresenta *il codice* della sorta.

```
Univ : Sort -> Type.
```

In seguito, si procede alla definizione della codifica dei termini utilizzando la definizione di `Term`. Questa funzione accetta una sorta, la sua codifica associata e restituisce il termine corrispondente.

```
def Term : s : Sort -> a : Univ s -> Type.
```

Viene poi definita `univ` con lo scopo di ritornare l'universo di una data sorta.

```
univ : s : Sort -> Univ (succ s).
```

### 3.3 I PROBLEMI DELL'EXPORT

#### 3.3.1 I PROBLEMI DI KRAJONO

Krajono presenta diverse problematiche, la principale delle quali è il fatto che non è più mantenuto da anni. È basato su una versione molto datata di Matita, che non solo manca delle funzionalità moderne, ma richiede anche versioni obsolete delle sue dipendenze. Ciò rende Krajono difficile da installare e far funzionare.

Un'altra problematica riguarda la modalità con cui l'utente avvia l'esportazione. Come evidenziato nella sezione 3.2.1, l'unico modo per avviarla è tramite il compilatore da linea di comando di Matita. Questa funzionalità non è integrata nell'interfaccia grafica di Matita, né supporta tutte le funzionalità fornite da essa.

Inoltre Krajono non esegue l'*escaping* delle *keyword* di Dedukti, quindi quando viene esportato un termine con una keyword di Dedukti come nome, si ottiene del codice non funzionante. Matita risolve questo problema aggiungendo un controllo adeguato durante l'esportazione e, nel caso venga rilevato un conflitto, genera un nuovo nome per il termine. In questo modo si evitano potenziali errori e conflitti durante l'esportazione dei termini.

Infine l'esportazione non tiene traccia delle dipendenze tra i vari moduli Dedukti processati. Di conseguenza i file esportati non hanno alcuna informazione che li lega alle loro dipendenze obbligando l'utente a ricostruirle manualmente.

#### 3.3.2 PERDITA DI INFORMAZIONE

**NORMALIZZAZIONE E CONFLUENZA** Dal punto di vista teorico, le problematiche dell'esportazione possono essere attribuite alla differenza tra il  $\lambda\Pi$ -calcolo modulo e il calcolo delle costruzioni (co)induttive, pertanto non sono risolvibili. In particolare, come evidenziato nella sezione 2.2.2, il  $\lambda\Pi$ -calcolo modulo non possiede le proprietà di normalizzazione e confluente che, al contrario, sono garantite dal calcolo delle costruzioni (co)induttive.



**PROOF IRRELEVANCE** La *proof irrelevance*, è un concetto nella teoria dei tipi che afferma che la prova di una proposizione o di un tipo non dovrebbe influire sul modo in cui questi vengono utilizzati. Ciò significa che due dimostrazioni diverse, di una stessa proposizione o tipo, possono essere considerate equivalenti e scambiate tra loro senza influire sulle conclusioni. In altre parole, una volta dimostrata una proprietà o un tipo, la specifica dimostrazione della sua validità diventa irrilevante. Grazie alla proof irrelevance si può semplificare l'utilizzo dei tipi e delle proposizioni dimostrate, poiché non è necessario considerare le specifiche dimostrazioni ogni volta, quando si ragiona su di essi.

Durante l'esportazione, le informazioni sulla proof irrelevance dei vari termini Matita vengono completamente perse.

**COSTRUTTI DI MATITA** Alcuni costrutti di Matita non hanno una traduzione diretta in Dedukti e pertanto vengono codificati come un blocco di termini. Durante la codifica perdono quindi la loro natura originale e con essa alcuni attributi chiave, senza i quali diventa impossibile ricostruire il termine di partenza. Questo rappresenta un problema che verrà affrontato nella sezione 4.3.



# 4 DA DEDUKTI A MATITA

In questo capitolo viene illustrato il funzionamento del processo di importazione del codice Dedukti in Matita. Prima viene discussa l'integrazione del parser di Dedukti, successivamente viene analizzata la strategia usata per la traduzione dei termini da un sistema all'altro. Infine viene trattata la ricostruzione di oggetti Matita precedentemente esportati.

## 4.1 DEDUKTI IN MATITA

Come visto nel capitolo 3, Kraiono consente l'esportazione del codice Matita verso Dedukti, tuttavia non è possibile fare il contrario. L'export è dunque a senso unico, in altre parole, qualcosa esportato da Kraiono non può essere re-importato. In seconda battuta, il progetto consiste nell'implementazione di una funzionalità di *import* che permetta a un utente di includere e utilizzare file Dedukti in Matita.

### 4.1.1 IL PARSER DEDUKTI

Per poter analizzare i file Dedukti, Matita sfrutta delle funzioni rese disponibili direttamente dal parser presente nel codice sorgente di Dedukti. Pertanto Dedukti è ora dipendenza di Matita. Tale parser è invocato dal motore di Matita durante il processo di compilazione, quando i file inclusi vengono processati istruzione per istruzione per poter costruire l'albero di sintassi astratta. Se uno di questi file è un sorgente Dedukti, allora il motore invoca il modulo OCaml responsabile dell'importazione. Questo chiama a sua volta il parser Dedukti che legge il contenuto del file e restituisce uno stream di entry. Que-

ste vengono analizzate nella loro struttura per poi costruire delle entry Matita equivalenti, da ritornare al motore.

## 4.2 L'IMPORT

### 4.2.1 LA GESTIONE DEI NOMI

Per gestire i nomi dei termini Matita fa uso di una particolare stringa contenente un *uri*. Tale stringa è in un formato che permette di capire in che file è stato definito, che tipo di termine rappresenta e la sua origine. Dedukti d'altra parte, identifica i suoi mediante una semplice coppia di stringhe composta dal nome del modulo Dedukti e da un identificatore proprio del termine.

È fondamentale per una corretta importazione tenere traccia di quali nomi sono assegnati a quale termine. A questo scopo Matita possiede una tabella hash in cui salva un riferimento al termine tradotto assieme al suo uri, costruito a partire dal nome del modulo Dedukti e l'identificatore del termine.

### 4.2.2 DEFINIZIONI E DICHIARAZIONI

**COSTANTI** Per tradurre le costanti viene semplicemente convertito il nome Dedukti in un uri Matita, controllando la tabella dei nomi per evitare conflitti.

**INDICI DI DE BRUJIN** Entrambi i software fanno uso degli indici di De Brujin, una rappresentazione compatta delle variabili legate all'interno di un termine. Sono utilizzati per semplificare la manipolazione dei termini, eliminando la necessità di utilizzare nomi unici per le variabili e consentendo di eseguire operazioni come la sostituzione e il confronto tra termini in modo efficiente. Essendo un indice rappresentato da un intero la conversione è diretta. L'unica accortezza è il sommare 1 a ciascun indice in quanto Dedukti fa uso di un sistema *0 based* mentre Matita conta partendo da 1.

**APPLICAZIONE** In Dedukti l'applicazione di una funzione è rappresentata tramite una tripla contenente:

- Un termine che rappresenta la funzione da applicare
- Un termine che rappresenta il primo argomento da passare alla funzione
- Una lista di termini che rappresenta il resto degli argomenti da passare

La traduzione consiste nel tradurre individualmente, ricorsivamente, ciascuno di questi termini, e assemblarli costruendo un oggetto che rappresenta l'applicazione in Matita.

$\lambda$ -ASTRAZIONE E PRODOTTO Astrazioni lambda e prodotti condividono la stessa struttura e pertanto sia in Dedukti che in Matita sono rappresentati in modo estremamente simile. Il primo usa una tripla contenente

- Un identificativo del termine
- Un termine rappresentante il tipo della  $\lambda$ -astrazione o del prodotto
- Un termine rappresentante il corpo

La traduzione consiste anche in questo caso nel creare un uri matita partendo dal identificativo di Dedukti e nel tradurre ricorsivamente il tipo e il corpo, per poi assemblare il tutto in un oggetto Matita.

TYPE E KIND Il  $\lambda\Pi$ -modulo usato da Dedukti rappresenta gli universi usando *type* e *kind*: il primo rappresenta l'insieme dei tipi dei termini, mentre il secondo rappresenta l'insieme dei tipi dei tipi. Ad esempio il numero 5 ha type *nat* e *nat* ha kind \*. Tuttavia, per come è definita la codifica utilizzata, questo tipo di termini non può comparire nel codice esportato, pertanto non li si traduce.

## 4.3 INVERTIRE L'ESPORTAZIONE

Durante la codifica applicata durante l'esportazione con Kraiono vengono perse informazioni su alcuni costrutti. Ad esempio, un tipo induttivo viene rappresentato attraverso un insieme di definizioni e regole di riscrittura, ma durante

questo processo si perde l'informazione che si tratta di un tipo induttivo. Pertanto se si vuole importare del codice Matita precedentemente esportato, non è sufficiente quanto descritto nella sezione precedente. Per poter ricostruire i termini Matita originali è necessario applicare qualche accorgimento.

##### 4.3.1 L'USO DELLE PRAGMA

Dedukti dà la possibilità all'utente di scrivere delle *direttive* o *pragma*. Queste sono delle righe di codice speciali interpretate solo dal compilatore e che non fanno parte del programma stesso. Attraverso l'uso di tali direttive, è possibile fornire istruzioni specifiche a Dedukti affinché agisca in modi particolari quando le incontra. Nel contesto di Matita, queste direttive vengono utilizzate per aggiungere informazioni durante l'esportazione in modo che, durante l'importazione, sia possibile ricostruire correttamente i termini.

La codifica di un termine Matita può risultare in un blocco di termini Dedukti, le pragma aggiunte durante l'esportazione forniscono informazioni sull'inizio e la fine di tale blocco. In aggiunta, al fine di ricostruire correttamente i termini, a volte è necessario conservare attributi aggiuntivi e specificare a quale termine si riferiscono. Di conseguenza le pragma contengono anche queste informazioni.

Le direttive vengono rappresentate internamente come stringhe di testo quindi, per semplificarne l'analisi, viene utilizzato uno standard specifico:

```
#PRAGMA [BEGIN|END] <NOME> [ATTR[:rif]=val]... .
```

Ad esempio

- `#PRAGMA FOO BAR=42 BAZ=43`. Significa: *L'istruzione successiva è di tipo FOO e possiede un attributo BAR con valore 42 e un attributo BAZ con valore 43.*
- `#PRAGMA BEGIN BLOCK GREETINGS:world=hello`. Significa: *Inizia un blocco di tipo BLOCK con attributo GREETINGS di valore hello facente riferimento a world.*
- `#PRAGMA END BLOCK`. Significa: *Fine di un blocco di tipo BLOCK.*

Un'alternativa a questo tipo di approccio avrebbe potuto prevedere l'utilizzo di pragma in cui gli attributi fossero memorizzati in modo rigido e non *human readable*. Questo da una parte avrebbe facilitato la scrittura del codice, ma dall'altra avrebbe reso difficoltosa la lettura e la scrittura di queste pragma. Inoltre, grazie all'aggiunta dei riferimenti sugli attributi, con questo approccio non si è vincolati a seguire un ordine rigido dei termini Dedukti all'interno di un blocco.

#### 4.3.2 COSTANTI CIC

Come descritto nella sezione 3.2.2, durante il processo di codifica, vengono utilizzate alcune definizioni di un modulo Dedukti chiamato "cic". Per ricostruire i termini esportati, la funzione responsabile della traduzione delle applicazioni si occupa anche di gestire ciascuna di queste definizioni in modo specifico, affrontando i casi uno per uno.

Nel caso di `cic.prod`, se viene applicata a un'astrazione con un corpo definito, si procede con la costruzione dell'oggetto `Matita` che rappresenta il prodotto. In caso contrario, si genera un errore poiché ciò risulta impossibile.

Nel caso di `cic.univ` e `cic.Univ`, viene costruito un oggetto `Matita` che rappresenta una sorta. Se vengono applicate a `cic.prop`, il tipo dell'oggetto `Matita` sarà "prop" (ovvero una proposizione). Se, invece, vengono applicate a `cic.type`, si procede a effettuare un conteggio ricorsivo del numero dell'universo.

#### 4.3.3 PUNTO FISSO

Un'istruzione di tipo punto fisso viene tradotta come un insieme di astrazioni lambda e regole di riscrittura. In particolare, per ogni punto fisso vengono create un'astrazione e una regola per rappresentare il tipo, e un'altra astrazione e un'altra regola per rappresentarne il corpo.

**LA PRAGMA** Per poter ricostruire l'oggetto `Matita` iniziale, oltre a individuare tipo e corpo, è necessario anche conoscere un parametro chiamato *reco*. Questo è un indice (*0-based*) che serve a individuare su quale argomento della

funzione avviene la ricorsione. Il valore però viene perso durante l'esportazione di Krajono. Per preservare queste informazioni durante l'esportazione, Matita aggiunge una coppia di pragma per delimitare per ciascun blocco di termini che rappresenta la codifica di un punto fisso. In tali direttive sono presenti anche gli attributi necessari alla ricostruzione. La pragma di apertura del blocco è nella forma:

```
#PRAGMA BEGIN FIXPOINT NAME=name RECNO:name=0.
```

Mentre la pragma di chiusura è nella forma:

```
#PRAGMA END FIXPOINT.
```

L'attributo `NAME` rappresenta il nome della funzione e può esserci più di un nome nel caso in cui si verifichi una ricorsione mutua. In altre parole, se diverse funzioni si richiamano a vicenda in modo ricorsivo, ciascuna di esse avrà un proprio attributo `NAME` che rappresenta il suo nome specifico. Questo permette di distinguere le diverse funzioni coinvolte nella ricorsione mutua e di gestirle correttamente durante la costruzione dell'oggetto Matita. Il *recno* viene esplicitato dall'omonimo attributo, specificando anche a quale funzione fa riferimento. Nel caso di ricorsione mutua, un esempio di pragma è:

```
#PRAGMA BEGIN FIXPOINT NAME=f NAME=g RECNO:f=0 RECNO:g=1.
```

**IL CORPO** La regola di riscrittura associata al corpo contiene il corpo del punto fisso a cui fa riferimento. Tuttavia non è possibile identificare una determinata regola di riscrittura in quanto non possiedono un identificativo di alcun tipo. Per capire quale regola tradurre Matita usa un ulteriore pragma:

```
#PRAGMA FIXPOINT_BODY REF=name.
```

Grazie all'attributo `REF`, è possibile ricostruire l'oggetto punto fisso originale in quanto memorizza a quale funzione fa riferimento il corpo.



## 4.3.4 TIPI INDUTTIVI

Ciascuna definizione di un tipo induttivo in Matita viene esportata usando una grande quantità di termini *Dedukti* e regole di riscrittura. In particolare vengono create:

- Una definizione che rappresenta il tipo del tipo induttivo.
- Per ogni costruttore del tipo, una definizione che rappresenta il tipo del costruttore.
- Una coppia di costanti e quattro regole di riscrittura per ogni universo e per *prop*. Queste fanno parte della codifica del pattern matching.
- Una lunga lista di costanti e regole di riscrittura. Queste rappresentano termini creati automaticamente dal runtime di Matita al momento della definizione del tipo induttivo e pertanto possono essere ignorati.

**LA PRAGMA** Per poter ricostruire un tipo induttivo Matita ha bisogno di poter risalire ai suoi costruttori e a un attributo intero chiamato *leftno* che rappresenta il numero di *parametri sinistri* del tipo induttivo. Anche in questo caso, al fine di preservare le informazioni che altrimenti andrebbero perse durante l'esportazione, Matita aggiunge una coppia di pragma per ogni blocco che rappresenta la codifica di un tipo induttivo. La pragma che apre il blocco presenta la seguente forma:

```
#PRAGMA BEGIN INDUCTIVE NAME=name LEFTNO=0 CONS:name=cons1
                                CONS:name=cons2.
```

Mentre quella di chiusura:

```
#PRAGMA END INDUCTIVE.
```

Per ogni tipo induttivo, vengono aggiunti un attributo **NAME** che indica il nome del tipo e una lista di attributi **CONS**, il cui valore corrisponde ai nomi dei vari costruttori, ciascuno con un riferimento al tipo associato. Viene aggiunto inoltre l'attributo **LEFTNO** per preservare il numero di parametri sinistri.

### 4.3.5 CODICE GENERATO

Come descritto nella sezione 4.3.4, in alcuni casi Matita genera automaticamente dei termini. Questi termini sono validi e vengono esportati. Tuttavia, è importante notare che non devono essere importati, poiché se si importano i termini che li hanno generati, Matita li ricreerà automaticamente.

Per indicare che un blocco rappresenta un termine che è stato generato automaticamente, Matita aggiunge una pragma prima e dopo il blocco durante la fase di esportazione. La pragma di apertura usata ha la seguente forma:

```
#PRAGMA BEGIN GENERATED.
```

mentre quella di chiusura:

```
#PRAGMA END GENERATED.
```

In questo modo, durante l'importazione, Matita è in grado di individuare i blocchi generati e di ignorarli.

# 5 CONCLUSIONI

## 5.1 CONCLUSIONI

Grazie a questo lavoro la funzionalità di esportazione presente in Krajono è disponibile anche sul Matita baseline. Oltre a ciò, è anche possibile importare delle dimostrazioni Dedukti in Matita. Nel caso in cui una dimostrazione rappresenti la codifica di una precedentemente esportata, vengono parzialmente ricostruiti i termini originali.

Anche se il lavoro svolto sull'importazione non ha coperto il caso del pattern matching, tutti gli altri casi sono stati largamente testati, prevedendo anche casi limite. La mancanza della traduzione del costrutto di pattern matching fa sì che certe definizioni che lo usano falliscano il controllo di tipo.

## 5.2 SVILUPPI FUTURI

Segue una lista di sviluppi possibili partendo da questo lavoro:

- *Pattern matching*: la funzionalità di importazione attualmente non gestisce il pattern matching. Per la corretta ricostruzione di questo costrutto è necessario definire una nuova pragma per salvare le informazioni di cui si ha bisogno. Successivamente serve processare le varie costanti che rappresentano il match presenti nell'encoding di ciascun tipo induttivo. Infine, quando traducendo un termine Dedukti si trova l'invocazione di una delle *costanti match*, bisogna costruire l'oggetto Matita, con tutti i suoi attributi.
- *Auto-inclusione dei file Dedukti*: attualmente una limitazione riscontrata è che l'esportazione di Krajono consente solo l'importazione in Matita

## 5 Conclusioni

di termini Dedukti definiti nello stesso modulo. Un'area di possibile sviluppo futuro potrebbe riguardare l'aggiunta di un'informazione nel file di output che consenta di identificare i file da importare in precedenza per evitare errori durante l'importazione del file corrente.

- *Integrazione con Matita*: per garantire una migliore esperienza utente nell'utilizzo delle funzionalità di importazione ed esportazione, sarebbe opportuno integrarle più strettamente con Matita. In particolare queste funzionalità dovrebbero essere integrate con le capacità di undo/redo per consentire l'annullamento o il ripristino delle modifiche effettuate e con le funzionalità che permettono di verificare una prova un passo alla volta. Inoltre, sarebbe importante integrarle con il meccanismo di serializzazione e deserializzazione utilizzato da Matita per il salvataggio dei file verificati. Infine sarebbe consigliabile fornire un pulsante nella UI che consenta l'esecuzione dell'esportazione, in modo da non dover usare sempre il compilatore da riga di comando.

# BIBLIOGRAFIA

- [1] abcdef.wiki. *Teorema di Church – Rosser - Church–Rosser theorem Teorema di Church– Rosser*. URL: [https://it.abcdef.wiki/wiki/Church%E2%80%93Rosser\\_theorem](https://it.abcdef.wiki/wiki/Church%E2%80%93Rosser_theorem).
- [2] A. Asperti, W. Ricciotti e C. Sacerdoti Coen. «Matita Tutorial». In: *Journal of Formalized Reasoning* 7.2 (gen. 2014), pp. 91–199. DOI: 10.6092/issn.1972-5787/4651. URL: <https://jfr.unibo.it/article/view/4651>.
- [3] A. Asperti, W. Ricciotti, C. Sacerdoti Coen e E. Tassi. «The Matita Interactive Theorem Prover». In: *Automated Deduction – CADE-23*. A cura di N. Bjørner e V. Sofronie-Stokkermans. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 64–69. ISBN: 978-3-642-22438-6.
- [4] D. Cousineau e G. Dowek. «Embedding Pure Type Systems in the Lambda-Pi-Calculus Modulo». In: *Typed Lambda Calculi and Applications*. A cura di S. R. Della Rocca. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 102–117. ISBN: 978-3-540-73228-0.
- [5] F. Thiré. «Sharing a Library between Proof Assistants: Reaching out to the HOL Family». In: *Electronic Proceedings in Theoretical Computer Science* 274 (lug. 2018), pp. 57–71. DOI: 10.4204/eptcs.274.5. URL: <https://doi.org/10.4204/eptcs.274.5>.



# RINGRAZIAMENTI

Questo è il coronamento simbolico di tre anni di sforzi e sacrifici miei, ma anche di chi mi è stato vicino.

Prima di tutto ringrazio Teodoro, Cristina, Martina, Chiara e tutto il resto della mia famiglia che, anche se fisicamente sono a 150 chilometri da Bologna, spiritualmente mi sono sempre stati vicini.

Ringrazio il Professor Claudio Sacerdoti Coen, mio relatore, per essere stato estremamente disponibile con me e per avermi fatto conoscere parte delle meraviglie che riserva la materia misteriosa e fantastica che è la Logica.

Ringrazio Clarissa, la mia supercoinquilina turing-completa, per essere una sicurezza e per avermi sopportato. Ti devo molto, questa laurea è anche tua.

Ringrazio i miei amici di su e di giù. Grazie per tutte le avventure che abbiamo vissuto in questi anni. Ringrazio Alvise, Andreea, Elena, Erik, Federica, Gabriele, Luca, Nicolò, Paolo B., Paolo C., Rickie, Stefano e tutti gli altri.

Infine ringrazio i miei colleghi del *Team Mobile* e dell'*Incubator* per avermi insegnato tantissimo durante questi anni. In particolare ringrazio Alexandru e Marco per avermi messo alla prova e avermi dato fiducia.