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

Indice

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

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 Basi teoriche

In questo capitolo vengono trattate le conoscenze teoriche necessarie per la comprensione del lavoro esposto in 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 trattati i *proof assistant* e il problema della loro interoperabilità.

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
- $falso$ 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 λ-termine del λ-calcolo di Church

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

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

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:

Operatori logici	**Tipi**
$\top$	Tipo unit
$\bot$	Tipo vuoto/void
$\wedge$	Tipi prodotto
$\vee$	Tipi somma
$\Rightarrow$	Tipi funzione
$\exists$	Tipi Σ
$\forall$	Tipi Π

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[1] 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 obbiettivi 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. Alcuni proof system, come ad esempio Coq e Hol Lite [2] infatti 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 λ-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

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

Mentre il sistema di tipi è definito dalle seguenti regole:

$$\frac{}{[\,]\ \textit{ben formato}}\ \textbf{Contesto vuoto}$$

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

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

[1] "Dedurre" in Esperanto
[2] Coq: https://github.com/Deducteam/CoqInE,
HOL Lite: https://arxiv.org/pdf/1507.08720.pdf

$$\frac{\Gamma \vdash \mathrm{t} \hookrightarrow \mathrm{u} \ \textit{ben formato}}{\Gamma,\ \mathrm{t} \hookrightarrow \mathrm{u} \ \textit{ben formato}}\ \textbf{Regola}$$

$$\frac{\Gamma \ \textit{ben formato}}{\Gamma \vdash \mathrm{Type} : \mathrm{Kind}}\ \textbf{Sorta}$$

$$\frac{\Gamma \ \textit{ben formato} \qquad \mathrm{x} : \mathrm{A} \in \Gamma}{\Gamma \vdash \mathrm{x} : \mathrm{A}}\ \textbf{Variabile}$$

$$\frac{\Gamma \vdash \mathrm{A} : \mathrm{Type} \qquad \Gamma,\ \mathrm{x} : \mathrm{A} \vdash \mathrm{B} : \mathrm{Type}}{\Gamma \vdash \Pi\ \mathrm{x} : \mathrm{A}\ .\ \mathrm{B} : \mathrm{Type}}\ \textbf{Prodotto (Type)}$$

$$\frac{\Gamma \vdash \mathrm{A} : \mathrm{Type} \qquad \Gamma,\ \mathrm{x} : \mathrm{A} \vdash \mathrm{B} : \mathrm{Kind}}{\Gamma \vdash \Pi\ \mathrm{x} : \mathrm{A}\ .\ \mathrm{B} : \mathrm{Kind}}\ \textbf{Prodotto (Kind)}$$

$$\frac{\Gamma \vdash \mathrm{A} : \mathrm{Type} \qquad \Gamma,\ \mathrm{x} : \mathrm{A} \vdash \mathrm{B} : \mathrm{Type} \qquad \Gamma,\ \mathrm{x} : \mathrm{A} \vdash \mathrm{t} : \mathrm{B}}{\Gamma \vdash \lambda\ \mathrm{x} : \mathrm{A}\ .\ \mathrm{t} : \Pi\ \mathrm{x} : \mathrm{A}\ .\ \mathrm{B}}\ \lambda\ \textbf{astrazione}$$

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

$$\frac{\Gamma \vdash \mathrm{A} : \mathrm{s} \qquad \Gamma \vdash \mathrm{x} : \mathrm{A} \qquad \mathrm{A} \equiv_{\beta\Gamma} \mathrm{B}}{\Gamma \vdash \mathrm{x} : \mathrm{B}}\ \textbf{Equivalenza}$$

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 λ-astrazioni e le applicazioni

$$\frac{\Gamma,\ \mathrm{x} : \mathrm{A} \vdash \mathrm{M} : \mathrm{T}}{\Gamma \vdash (\lambda \mathrm{x} : \mathrm{A}\ .\ \mathrm{M}) : \mathrm{A} \to \mathrm{B}}\ \lambda\textbf{-astrazione}$$

$$\frac{\Gamma \vdash \mathrm{f} : \mathrm{A} \to \mathrm{B} \qquad \Gamma \vdash \mathrm{x} : \mathrm{A}}{\Gamma \vdash (f\ x) : \mathrm{B}}\ \textbf{Applicazione}$$

CIC è più ricco del λΠ modulo usato da Dedukti in quanto 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] ≝
   E : list
 | C : nat → list → list.
```

I punti fissi sono utili per definire funzioni ricorsive, mentre l'aggiunta del pattern matching permette 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 ≝
 match l with
 [ E ⇒ 0
 | C hd tl ⇒ 1 + len tl
 ].
```

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 [1] [2]. 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, il 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*Matita with embedded Elpi*[3].

[1] Significa "Matita" in Esperanto
[2] https://github.com/Deducteam/Krajono
[3] 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* contenente il codice matita esportato.

Quando il flag menzionato è attivato, 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 esporazione non è sempre un encoding *uno a uno*, ma determinati tipologie di termini Matita vengono codificati in più termini Dedukti.

3.2.2 L'encoding

kdjsaoijdisa

3.3 I problemi di Krajono

4 Da Dedukti a Matita

In questo capitolo viene illustrato il funzionamento del processo di importazione del codice Dedukti in Matita. Prima

4.1 Dedukti in Matita

Come visto nel capitolo 3, Krajono consente l'esportazione del codice Matita verso Dedukti, tuttavia non è possibile fare il contrario. L'export è dunque a senso unico e qualcosa esportato da Krajono 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 il 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 invoca a sua volta il parser Dedukti che legge il contenuto del file e restituisce uno stream di entry. Queste vengono analizzate nella loro struttura per poi costruire delle entry Matità equivalenti, da poi ritornare al motore.

4.2 L'import

Per la maggior parte dei termini Dedukti esiste una controparte equivalente in Matita. Dunque il processo di importazione si riduce al convertire un oggetto in un altro mettendo le informazioni giuste al posto giusto.

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 stata 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 efficente. 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.

λ-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. Nel primo sono rappresentati come una tripla contenente

- Un identificativo del termine
- Un termine rappresentante il tipo della λ-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 rappresente l'insieme dei tipi dei tipi. Ad esempio il numero 5 ha type *nat* e *nat* ha kind $*$. (TODO)

4.3 Invertire l'esportazione

Durante la codifica applicata durante l'esportazione con Krajono 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 sufficente 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 da 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 direttive pragma 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, e 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 posside 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`.*

4.3.2 Costanti cic

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

Nel caso della costante `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 delle costanti `cic.univ` e `cic.Univ`, viene costruito un oggetto Matita che rappresenta una sorta. Se vengono applicate a una costante `cic.prop`, il tipo dell'oggetto Matita sarà "prop" (ovvero una sorta di tipo proposizionale). Se, invece, vengono applicate a una costante `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 *recno*. 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 puntofisso 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

- 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 per tanto 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 GENRATED.
```

mentre quella di chiusura:

```
#PRAGMA END GENRATED.
```

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

5 Conclusioni & sviluppi futuri

aisdjsfjd

Bibliografia

[1] 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.

[2] 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.

[3] 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.