

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

Sommario

Indice

1	Basi teoriche	4
1.1	La Teoria dei Tipi	4
1.1.1	La Teoria dei Tipi	4
1.1.2	L'isomorfismo di Curry-Howard	5
1.1.3	La teoria dei tipi nella pratica	5
1.2	Dedukti e Matita	5
1.2.1	Dimostratori Interattivi di Teoremi	5
1.2.2	Dedukti	6
1.2.3	Matita	8
1.3	Interoperabilità Dedukti-Matita	9
2	Da Matita a Dedukti	10
2.1	Krajono	10
2.1.1	Funzionamento dell'esportazione	10
2.1.2	Export da linea di comando	10
2.1.3	Integrazione in Matita	11
2.2	Problemi di Krajono	11
3	Da Dedukti a Matita	12
3.1	Il parser Dedukti	12
3.2	Tradurre i termini Dedukti	12
3.3	Invertire l'esportazione	13
3.3.1	L'uso delle pragma	14
3.3.2	Punto fisso	15
3.3.3	Tipi induttivi	15
3.3.4	pattern matching	15
3.3.5	Pragma generated	15
4	Conclusioni	16
5	Sviluppi futuri	17

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 ad una frammentazione in quanto sistemi diversi, che usano logiche diverse, non sono compatibili tra loro. Per ovviare a questo problema un gruppo di ricercatori del INRIA ha creato *Dedukti*, la cui idea è quella di agire da *type checker* per definizioni, enunciati e prove codificati nel calcolo da lui implementato. 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 calcoli. Grazie ad 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*, un proof assistant in sviluppo nel Dipartimento di Informatica 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 nel implementare la possibilità di re-importare 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 è più rigido e strutturato del metalinguaggio di Dedukti, pertanto durante l'esportazione venivano 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 l'encoding siano confluenti e normalizzanti. Dedukti non effettua questa verifica, che demanda ad altri tool. 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 l'encoding in Dedukti, nonostante le regole fossero anche confluenti e normalizzanti.

Capitolo 1

Basi teoriche

In questo capitolo vengono affrontate le conoscenze teoriche necessarie per la comprensione del lavoro esposto nei capitoli seguenti. Nella sezione 1.1 vengono illustrati i fondamenti della *Teoria dei Tipi*, teoria alla base di questa tesi. Nella sezione 1.2 vengono trattati i *proof assistant* e il problema della loro interoperabilità.

1.1 La Teoria dei Tipi

1.1.1 La Teoria dei Tipi

La *teoria dei tipi* è una branca della logica e dell'informatica teorica il cui obbiettivo è quello di studiare i così detti *type system*, ovvero insiemi di regole che associano una proprietà chiamata *tipo* a degli oggetti chiamati *termini*. Intuitivamente, assegnare un tipo ad 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 $':'$.

Nella teoria dei tipi, anche le funzioni sono termini e possono essere a loro volta tipizzate. Ad esempio, la seguente funzione rappresentata con un λ -termine appartenente al λ -calcolo di Church

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

è definita da \mathbb{N} a \mathbb{N} , e pertanto ha tipo $\mathbb{N} \rightarrow \mathbb{N}$

1.1.2 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 è 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:

Logica	Informatica
\top	Tipo unit
\perp	Tipo vuoto/void
\wedge	Tipi prodotto
\vee	Tipi somma
\Rightarrow	Tipi funzione
\exists	Tipi Σ
\forall	Tipi Π

1.1.3 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, grazie all'isomorfismo di Curry-Howard, ha permesso lo sviluppo di dimostratori interattivi di teoremi, i quali sono soggetto di questa tesi.

1.2 Dedukti e Matita

1.2.1 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, simile ad un linguaggio di programmazione, 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 attraverso un linguaggio di programmazione dotato di un sistema di tipi avanzato, ottenendo alla fine l'equivalente di 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 porta sicuramente un beneficio alla 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 ad una *frammentazione*. 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.

1.2.2 Dedukti

Dedukti¹ 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 quindi è creare una connessione tra i diversi sistemi di dimostrazione assistita. Ciò significa che le dimostrazioni possono essere tradotte da un sistema all'altro, agevolando lo scambio e il riutilizzo delle dimostrazioni 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.

λ I-Calcolo modulo Alla base di questo logical framework c'è il λ I-calcolo modulo (o semplicemente λ I modulo), un'estensione del λ -calcolo che introduce la *tipizzazione dipendente* e le *regole di riscrittura*. La prima

¹"dedurre" in esperanto

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

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

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}$$

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

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

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

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

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

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

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

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

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

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

1.2.3 Matita

Matita è un proof assistant in sviluppo nel dipartimento di informatica 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 auto-referenzianti. Le regole di tipaggio del CIC per i tipi semplici sono quelle comuni dei linguaggi di programmazione. Di seguito sono riportate quelle per le astrazioni e le applicazioni

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

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

CIC è più ricco del $\lambda\Pi$ modulo usato da Dedukti in quanto gode di *tipi induttivi*, *punti fissi* e *pattern matching*. I tipi induttivi consentono la creazione di tipi complessi e autoreferenziali. Possono essere utilizzati per definire strutture matematiche illimitate come liste, alberi o grafi. L'esempio seguente mostra come si può definire una lista concatenata di numeri naturali usando Matita. Una lista 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.
```

Invece punti fissi e il pattern matching sono utili per definire funzioni ricorsive strutturali. Il primo ci permette di definire funzioni ricorsive, mentre il secondo ci permette di ragionare per casi sulla struttura di un termine. Ad esempio, se volessimo calcolare la lunghezza di una `list` dell'esempio precedente si potrebbero fare delle considerazioni sulla sua struttura del tipo: *qual è la lunghezza della lista vuota? Qual è la lunghezza di una lista formata da un numero naturale e un'altra lista?*. Il seguente codice Matita mostra una possibile soluzione al problema di calcolare la lunghezza di una lista:

```
let rec len l on l =
  match l with
  [ E      0
  | C hd tl 1 + len tl
  ].
```

1.3 Interoperabilità Dedukti-Matita

Capitolo 2

Da Matita a Dedukti

2.1 Krajono

Attorno al 2018 un team di sviluppatori del *Institut national de recherche en informatique et en automatique* ha sviluppato un fork di Matita con la possibilità di esportare le dimostrazioni in un formato compatibile con Dedukti. Questo fork è tutt'ora distribuito pubblicamente con il nome di *Krajono* ("matita" in esperanto) anche se non è stato aggiornato con gli ultimi sviluppi del Matita baseline. Il primo passo del mio lavoro è stato quello di integrare il codice di Krajono dentro a Matita.

2.1.1 Funzionamento dell'esportazione

Il processo di esportazione prevede l'analisi del tipo e della struttura di ciascuna istruzione del file Matita. Queste passano per un processo di *scanning* e *parsing*, fino a diventare oggetti contenenti i termini, in un albero astratto. Questi vengono poi passati singolarmente al modulo responsabile per la conversione. Durante questo processo vengono analizzate la struttura e le caratteristiche di ciascun termine, per poi costruirne uno o più rappresentanti istruzioni Dedukti. La traduzione cerca di essere dove possibile una trasformazione uno a uno, anche se come si vedrà nella sezione 2.2 non sempre è possibile.

2.1.2 Export da linea di comando

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. Opzionalmente, passando come argomento il nome di un file, è possibile anche compilarlo singolarmente.

Krajono fornisce la possibilità di esportare il codice Matita specificando l'argomento `-extract_dedukti` a Matitac, sia lavorando con un unico file, sia con un'intera directory. Successivamente si potranno trovare i file `.dk` relativi al codice matita esportato nella directory dei sorgenti.

2.1.3 Integrazione in Matita

Per integrare le funzionalità di Krajono in Matita al posto di aprire una *pull request* da una repository Git all'altra, si è preferito copiare e adattare i singoli file sorgente responsabili dell'esportazione. Questo in quanto utilizzare lo strumento di versionamento automatico avrebbe portato delle complicazioni in quanto Krajono non è basato direttamente sul Matita baseline, ma su un ulteriore fork ¹. Inoltre Krajono, al momento della stesura di questa tesi, non è più mantenuto da anni.

Struttura del sorgente In codice sorgente responsabile dell'esportazione è diviso in tre moduli OCaml:

- *dedukti*: contenente le definizioni di costanti e funzioni di utilità.
- *deduktiExtraction*: contenente la logica del vero e proprio export.
- *deduktiPrint*: contenente semplici funzioni per stampare a schermo gli oggetti usati nell'export.

Durante la compilazione di un file, quando il flag `-extract_dedukti` è attivo, il motore di Matita/Krajono avvia l'esportazione chiamando una funzione dal modulo `deduktiExtraction` durante la fase di costruzione dell'albero astratto.

Integrazione Il codice interessato dunque è poco dipendente dal resto del codice di Matita, quindi il porting della funzionalità si è ridotto al copiare i file da un progetto all'altro e modificare dove necessario. Ora è possibile avviare l'export tramite lo stesso flag di Matitac presente in Krajono.

2.2 Problemi di Krajono

Essendo il linguaggio di Dedukti meno espressivo del

¹<https://github.com/LPCIC/matita>

Capitolo 3

Da Dedukti a Matita

Come visto nel capitolo precedente, usando Krajono è possibile esportare del codice Matita verso Dedukti, tuttavia non è possibile fare il contrario, in quanto ne Krajono, ne Dedukti stesso godono di questa funzionalità. L'export è dunque a senso unico, e qualcosa di esportato non può essere re-importato in Matita. Il lavoro di questa tesi è proprio il seguente: rendere Matita capace di esportare ed importare codice da e verso Dedukti. Con un export a doppio senso gli sviluppatori Matita saranno in grado di usare dimostrazioni Dedukti e vice versa.

3.1 Il parser Dedukti

3.2 Tradurre i termini Dedukti

Alcuni termini Dedukti sono direttamente traducibili in termini Matita. Alcuni invece richiedono una logica più complessa.

Costanti Per tradurre le costanti è stato necessario semplicemente convertire il nome Dedukti in un *uri* Matita. Per evitare conflitti con i nomi si è aggiunto anche una tabella hash che per tenere traccia dei nomi già assegnati e dei relativi uri.

Indici di De Bruijn Entrambi i software fanno uso degli indici di De Bruijn, 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

stata diretta. L'unica accortezza presa è stata l'aggiungere 1 ad ogni indice in quanto Matita fa uso di un sistema 1 *based* mentre Dedukti conta partendo da 0.

β -riduzione In Dedukti il passo di β -riduzione è rappresentato tramite una tripla contenente:

- Un termine rappresentante la λ astrazione da ridurre
- Un termine rappresentante il primo argomento da usare per la riduzione
- Una lista di termini rappresentante il resto degli argomenti

Per tradurlo è stato sufficiente tradurre individualmente, ricorsivamente, ciascuno di questi termini, e assemblarli costruendo un oggetto che rappresenta la β -riduzione in Matita. L'esportazione del capitolo 2 utilizza delle particolari definizioni per rappresentare alcuni termini di Matita

λ -astrazione e prodotto Astrazioni lambda e prodotti condividono la stessa struttura e pertanto sono rappresentati allo stesso modo sia in Dedukti che in Matita. Nel primo sono rappresentati come una tripla contenente

- Un identificativo Dedukti
- Un termine rappresentante il tipo della λ -astrazione o del prodotto
- Un termine rappresentante il corpo

Si costruiscono quindi i relativi oggetti Matita convertendo l'identificativo e traducendo ricorsivamente il tipo e il corpo.

Type e Kind Il calcolo lambda-pi usa i concetti di *type* e *kind*: *type* è il classico *concetto di tipo* usato per la classificazione di termini, mentre *kind* è un tipo speciale rappresentante il tipo di tutti i tipi. Ad esempio 5 può avere tipo *nat*, mentre *nat* TODO avere kind ***. Nel calcolo delle costruzioni (co)induttive alla base di Matita questi concetti non sono presenti, e pertanto non sono stati tradotti.

3.3 Invertire l'esportazione

Fino adesso è stato visto come importare in Matita del codice Dedukti semplice, tuttavia, se si volesse importare del codice precedentemente esportato

usando la funzionalità del capitolo 2, ci si accorgerebbe della scomparsa di alcuni costrutti Matita, come ad esempio il *match*. Questo perché, dato che Dedukti non li possiede, durante l'esportazione sono stati trasformati in termini che ne emulano il comportamento. Per rendere dunque possibile la costruzione di un codice quanto più vicino all'originale si ha pensato ed implementato la strategia qua successivamente discussa.

3.3.1 L'uso delle pragma

Il linguaggio di Dedukti dà la possibilità all'utente di scrivere delle *direttive* o *pragma*. Queste sono delle particolari righe di codice interpretate dal compilatore e che pertanto non fanno parte del programma. Usandole è possibile istruire il compilatore Dedukti, o nel nostro caso il parser Dedukti integrato dentro Matita 3.1, affinché agisca in determinati modi quando le incontra. Nel caso specifico di questa tesi, sono state definite ed usate delle pragma per indicare quali parti di codice Dedukti che fanno riferimento ad un costrutto Matita andato perso durante l'esportazione.

Sintassi Le pragma sono sostanzialmente delle stringhe, quindi è stato necessario pensare ad uno standard che aiutasse a strutturarle e ne facilitasse il parsing. Dato che la traduzione di un costrutto Matita in Dedukti può risultare in un blocco di istruzioni bisogna essere in grado di capire dove questo inizia e finisce. Inoltre, per poter ricostruire un oggetto talvolta è necessario salvare degli attributi aggiuntivi, specificando anche a quale oggetto fanno riferimento. Per tanto si è pensato di inserire pragma con la seguente sintassi:

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

Le seguenti pragma ad esempio sono valide

- `#PRAGMA FOO BAR=42`. L'istruzione successiva è di tipo `FOO` e ha l'attributo `BAR` che vale 42.
- `#PRAGMA BEGIN BLOCK GREETINGS:world=hello`. Inizia un blocco di tipo `BLOCK` con l'attributo `GREETINGS` di valore `hello` che fa riferimento a `world`.
- `#PRAGMA END BLOCK`. Fine di un blocco di tipo `BLOCK`

3.3.2 Punto fisso

Dato un insieme A e una funzione $f : A \rightarrow A$, $x \in A$ si dice *punto fisso* di f se e solo se $x = f(x)$. Nella teoria dei tipi questo concetto è utile per rappresentare le funzioni ricorsive. Questo è uno di quei costrutti che viene perso durante l'esportazione, in quanto Matita ne fa uso mentre Dedukti no.

Punto fisso nell'encoding Un'istruzione di tipo punto fisso è tradotta come un insieme di astrazioni lambda e delle regole di riscrittura. In particolare per ciascun punto fisso si ottengono un'astrazione e una regola per rappresentare il tipo e un'astrazione e una regola per rappresentare il corpo.

La pragma Per poter ricostruire l'oggetto Matita iniziale, oltre ad individuare tipo e corpo, è necessario anche conoscere un parametro chiamato *recno*. Questo è un indice (*0-based*) che serve ad individuare su quale argomento della funzione avviene la ricorsione. Il valore però viene anch'esso perso durante l'esportazione, quindi è stato necessario trovare un modo per preservarlo. Per conservare queste informazioni si è modificato il codice dell'esportazione in modo da inserire delle pragma come delimitatori del blocco di istruzioni Dedukti rappresentanti l'encoding del fixpoint. In particolare la pragma per indicare l'inizio 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 ce ne possono essere più di uno nel caso in cui si abbia della ricorsione mutua. Il *recno* viene esplicitato nell'omonimo attributo, specificando anche a quale nome fa riferimento. Nel caso di ricorsione mutua si potrebbe ottenere una pragma nella forma:

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

Il corpo

3.3.3 Tipi induttivi

3.3.4 pattern matching

3.3.5 Pragma generated

Capitolo 4

Conclusioni

Capitolo 5

Sviluppi futuri