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

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 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 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*, 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 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 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 trattate le conoscenze teoriche necessarie per la comprensione del lavoro esposto nei capitoli successivi. Nella sezione 1.1 sono 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 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 ':'.

Nella 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 $\mathbb{N} \to \mathbb{N}$

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 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

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 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 a 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

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 espres-

[1] "dedurre" in esperanto

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

sioni, 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:

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

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

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

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

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

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

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

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

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

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

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

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 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

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

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

CIC è più ricco del $\lambda\Pi$ modulo usato da Dedukti in quanto consente di definire *tipi induttivi*, *punti fissi* e *pattern matching*. I primi 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 i punti fissi sono utili per definire funzioni ricorsive e, con l'aggiunta del pattern matching, definiscono funzioni ricorsive strutturali.

Il seguente esempio mostra l'implementazione di una funzione che calcola la lunghezza di una `list` definita nell'esempio precedente. La funzione ragiona sulla struttura della lista usando il pattern matching.

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

Capitolo 2

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.

2.1 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 attualmente questo fork non è più mantenuto da anni.

2.1.1 Integrazione in Matita

In prima battuta, il progetto consiste in un'integrazione delle funzionalità di export di Krajono nel Matita baseline. Tali funzionalità sono presenti in Matita grazie alla copia dei singoli sorgenti responsabili dell'esportazione.

Le funzionalità non vengono integrate mediante *pull request* da una repository *Git* all'altra in quanto comporterebbe delle complicazioni visto che Krajono non solo non è mantenuto da anni, ma è basato ma su un ulteriore fork di Matita, chiamato *Matita with embedded Elpi*[3].

[1] Significa "Matita" in Esperanto
[2] https://github.com/Deducteam/Krajono
[3] https://github.com/LPCIC/matita

Il codice sorgente responsabile dell'esportazione è diviso in pochi moduli OCaml quasi del tutto indipendenti dal restante codice sorgente di Matita, per tanto integrarli nel Matita moderno non causa problemi.

2.2 L'export

2.2.1 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. Opzionalmente, passando come argomento il nome di un file, è possibile anche compilare solo quello singolarmente. 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 suddetto flag è attivo il motore di Matita avvia l'esportazione chiamando una funzione di 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.

2.2.2 L'encoding

2.3 Problemi di Krajono

Capitolo 3

Da Dedukti a Matita

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

3.1 Dedukti in Matita

Come visto nel capitolo 2, 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 ad un utente di includere e utilizzare file Dedukti in Matita.

3.1.1 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 Matita equivalenti, da poi ritornare al motore.

3.2 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.

3.2.1 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.

3.2.2 Definizioni e Dichiarazioni

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 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 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 da 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 **??**, affinché agisca in determinati modi quando le incontra. Nel caso specifico di questa tesi, sono state definite e usate delle pragma per indicare quali parti di codice Dedukti che fanno riferimento a un costrutto Matita andato perso durante l'esportazione.

Sintassi Le pragma sono sostanzialmente delle stringhe, quindi è stato necessario pensare a 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 $f : A \to A$, $x \in A$ 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 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 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 & sviluppi futuri