Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

LA MIA FANTASTICA OTTIMISTICA TESI

Relatore:
Chiar.mo Prof.
Claudio Sacerdoti Coen

Presentata da:
Mattia Girolimetto

I Appello di Laurea
Anno Accademico 2022-2023

Sommario

Indice

Capitolo 1

Introduzione

Capitolo 2

Proof Assistant e Interoperabilità

2.1 La Teoria dei Tipi

2.1.1 Il paradosso di Russel

Quando il matematico inglese Bertrand Russel propose il paradosso oggi chiamato a suo nome, nei primi anni del '900 scatenò quella che venne definita *crisi dei fondamenti matematici*. Il paradosso semplicemente evidenziava come la teoria degli insiemi usata fino a quel punto (oggi definita teoria degli insiemi *naive*) rendesse possibile definire un insieme come il seguente

$$X = \{Y | Y \notin Y\}$$

La contraddizione avviene quando ci si domanda se $X \in X$, in quanto se X appartenese a sé stesso allora non apparterrebbe all'insieme degli insiemi che appartengono loro stessi, ovvero X stesso. In formule:

$$X \in X \Leftrightarrow X \notin X$$

Una delle strategie pensate dunque per aggirare questo paradosso fu la *teoria dei tipi*.

2.1.2 La Teoria dei Tipi

La *teoria dei tipi* è una branca della matematica, della logica, 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* ad degli oggetti

chiamati *termini*. Nonostante siano state proposte molteplici teorie, le principali emerse sono due: il *λ-calcolo tipato* di Alonozo Church e la *teoria dei tipi intuizionistica* di Per Martin-Löf.

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 ':'. Gli esempi precedenti diventano quindi:

- $42 : \mathbb{N}$
- $-5 : \mathbb{Z}$
- $falso : \mathbb{B}$

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 per tanto ha tipo $\mathbb{N} \to \mathbb{N}$

2.1.3 L'isomorfismo di Curry-Howard

Sempre 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 analogo al verificare la correttezza degli assegnamenti di tipo di un programma (*type checking*). Nella sua formulazione più generale, l'isomorfismo di Curry-Howard può essere riassunto con la seguente tabella:

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

2.1.4 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'isomorifismo di Curry-Howard, ha permesso lo sviluppo di dimostratori automatici di teoremi e di dimostratori interattivi di teoremi, i quali sono soggetto di questa tesi.

2.2 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 corretezza 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.

2.2.1 Matita

Matita è un proof assistant in sviluppo nel dipartimento di informatica dell' Università di Bologna. E' open source, scritto nel linguaggio di programmazione OCaml ed è rilasciato secondo i termini della GNU General Public Licence. E' basato sul *calcolo delle costruzioni (co)induttive*, una teoria di tipi dipendenti che estende il *calcolo delle costruzioni* sviluppato da Thierry Coquand aggiungendo i tipi induttivi, ovvero tipi autoreferenzianti.

2.2.2 Dedukti

Dedukti[1] è un *logical framework* sviluppato da alcuni ricercatori del *Institut national de recherche en informatique et en automatique* francese. Il software è open source, anch'esso 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 al computer. 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] godono infatti della possibilità di esportare e importare codice da e verso Dedukti.

Si basa sul $\lambda\pi$-calcolo, un'estensione del λ-calcolo che introduce la tipizzazione dipendente, consentendo la specifica di tipi complessi che dipendono dai valori delle espressioni. Questa caratteristica lo rende un potente strumento per la verifica formale e la dimostrazione assistita tramite proof assistant.

2.3 Interoperabilità Dedukti-Matita

2.3.1 Interoperabilità tra sistemi

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, unito alla forte diversità che li caratterizza individualmente, questo fenomeno porta inevitabilmente ad una *frammentazione* della conoscenza. Non è quasi mai possibile infatti per un utente dimostrare la veridicità di un teorema usando un proof assistant e usare la stessa dimostrazione in un altro di questi tool. Il problema è dovuto a fattori facilmente aggirabili, come ad esempio la differenza sintattica dei due linguaggi proprietari, ma anche a fattori non facilmente aggirarabili, come nel caso in cui i due tool usino calcoli con diversi livelli di espressività.

Nasce dunque l'esigenza di favorire l'interoperabilità tra questi sistemi, in modo da arginare questo problema e favorire lo sviluppo scientifico. A tale scopo, nel tempo sono state aggiunte ad alcuni di tool delle funzionalità di export, per permettere all'utente di ottenere la propria dimostrazione in un formato compatibile con un altro software.

[1] "dedurre" in esperanto

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

Capitolo 3

Da Matita a Dedukti

3.1 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.

3.1.1 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 vegono 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 3.2 non sempre è possibile.

3.1.2 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.

3.1.3 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 addattare 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.

3.2 Problemi di Krajono

Essendo il linguaggio di Dedukti meno espressivo del

[1] https://github.com/LPCIC/matita

Capitolo 4

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 reimportato 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.

4.1 Il parser Dedukti

4.2 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 Brujin Entrambi i software fanno uso degli indici di De Brujin, una rappresentazione compatta delle variabili legateall'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 sufficente tradurre individualmente, ricorsivamente, ciasucno di questi termini, e riassemblarli costruiendo un oggetto che rappresenta la β-riduzione in Matita. L'esportazione del capitolo 3 utilizza delle particolari definizioni per rappresentare alcuni termini di Matita

λ-astrazione e prodotto Astrazioni lambda e prodotti condividono la stessa struttura e per tanto sono rappresentati allo stesso modo sia in Dedukti che in Matita. Nel primo sono rappresentati come una tupla 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* porebbe avere kind $*$. Nel calcolo delle costruzioni (co)induttive alla base di Matita questi concetti non sono presenti, e per tanto non sono stati tradotti.

4.3 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 3, 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.

4.3.1 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 per tanto non fanno parte del programma. Usandole è possibile istruire il compilatore Dedukti, o nel nostro caso il parser Dedukti integrato dentro Matita 4.1, 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.

Sinstassi Le pragma sono sostanzialmente delle stringhe, quindi è stato necessario pensare ad uno standard che aiutasse a strutturarle e ne faclitasse 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`

4.3.2 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 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 indiciare l'inizio del blocco è nella forma:

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

Mentre la pragma di chisura 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

4.3.3 Tipi induttivi

4.3.4 pattern matching

4.3.5 Pragma generated

Capitolo 5

Conclusioni

Capitolo 6

Sviluppi futuri