Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

Implementazione del linguaggio dichiarativo in Matita 0.99.x

Relatore:
Chiar.mo Prof.
Claudio Sacerdoti Coen

Presentata da:
Andrea Berlingieri

II Sessione - primo appello
Anno Accademico 2018/2019

Indice

Motivazioni

In questo lavoro di tesi verrà presentato il porting del linguaggio dichiarativo del dimostratore interattivo di teoremi Matita dalla versione 0.5.x alla versione 0.99.x.

Le motivazioni che stanno dietro alla scelta di un lavoro di questo tipo sono varie. Innanzitutto c'è il grande interesse per la Logica Matematica, con cui sono venuto a contatto durante il primo anno del corso di Laurea e di cui mi sono presto appassionato. Questo lavoro mi ha dato quindi l'opportunità di approfondire alcuni aspetti della materia che non fanno parte del programma del corso.

Un altro fattore trainante nella scelta è stata la curiosità riguardo i meccanismi alla base di Matita, un software con cui ho avuto a che fare durante il corso di Logica. Alcuni dei quesiti a cui ho cercato risposta sono stati come facesse uno strumento software a garantire la correttezza di una dimostrazione leggendone uno script in linguaggio quasi naturale, e quali fossero i limiti dei calcolatori nell'utilizzo come supporto alle dimostrazioni formali.

Infine, uno dei motivi più importanti che mi hanno portato lungo questo percorso è stata la forte convinzione che strumenti software come Matita abbiano un'immensa utilità dal punto di vista didattico. Ho apprezzato molto la possibilità di mettere in pratica le nozioni teoriche apprese con un programma che garantisse che la mia comprensione fosse corretta. Parte fondamentale dell'apprendimento della materia è senz'altro la messa in pratica delle più tipiche tecniche di dimostrazione formale. Con Matita questa parte di pratica è stata agevolata sia per la possibilità di svolgere esercizi di dimostrazione formale in completa autonomia, sia per la possibilità di avere un feedback istantaneo rispetto alla correttezza degli esercizi, senza la necessità di chiedere colloquio al professore.

Quello che ho voluto fare con questo lavoro è stato portare un contributo, seppur

modesto, ad un progetto che credo sia molto importante per la didattica di una materia notoriamente ardua come la Logica, nella speranza che i futuri studenti di Informatica a Bologna possano apprezzare un Matita migliorato durante il loro percorso di studi.

Elenco delle figure

Capitolo 1

Introduzione

1.1 Interactive theorem provers

Un interactive theorem prover è uno strumento software per il supporto alle dimostrazioni formali. Molti interactive theorem prover funzionano come dei "gestori" di prove, permettendo di costruirne interagendo con lo strumento software mediante un'interfaccia utente, tipicamente testuale. Le prove sono poi memorizzate in un formato interno. Esempi di interactive theorem prover sono Coq [1], Mizar [4], NuPRL [5], e Matita [3].

Gli interactive theorem prover sono utilizzati per la dimostrazione formale di correttezza di sistemi software, nell'ambito, ad esempio, di processi di sviluppo formali. Un altro utilizzo possibile, che viene fatto all'università Bologna, è quello di supporto alla didattica della Logica Matematica.

Tutti gli ITP, alla loro base, hanno un lambda calcolo tipato. Nel caso di Coq e Matita, questo calcolo è il Calcolo delle Costruzioni (Co)induttive, ispirato dai lavori di Per Martin Löf [14] e Jean-Yves Girard [13]. Il principio base degli ITP basati su lambda calcoli tipati, che mette in relazione un formalismo di calcolo ed una logica, è l'isomorfismo di Curry-Howard. Lo si può esprimere, in maniera informale, nel seguente modo: per alcune coppie composte da un calcolo logico ed un lambda calcolo tipato esiste un isomorfismo che assegna, ad ogni "aspetto" del calcolo logico, un "aspetto" del lambda calcolo tipato.

Per rendere l'idea con un semplice esempio, consideriamo il lambda calcolo tipato i

cui termini sono definiti dalla seguente grammatica:

$$\boldsymbol{v} ::= x \mid y \mid z \mid \ldots$$

$$\boldsymbol{t} ::= \boldsymbol{v} \mid \boldsymbol{t}\ \boldsymbol{t} \mid \lambda \boldsymbol{v}.\boldsymbol{t} \mid \langle \boldsymbol{t}, \boldsymbol{t} \rangle \mid \pi_1 \boldsymbol{t} \mid \pi_2 \boldsymbol{t}$$

Ad ogni termine assegnamo un tipo, con la sintassi $t : T$. I tipi sono definiti dalla seguente grammatica:

$$\boldsymbol{V} ::= \boldsymbol{A} \mid \boldsymbol{B} \mid \ldots$$

$$\boldsymbol{T} ::= \boldsymbol{V} \mid \boldsymbol{T} \times \boldsymbol{T} \mid \boldsymbol{T} \to \boldsymbol{T}$$

Abbiamo quindi dei tipi "di base", e dei costruttori di tipo per costruire nuovi tipi a partire da quelli esistenti.

Al calcolo aggiungiamo le seguenti regole di riscrittura:

$$(\lambda x.t)v \to t[v/x] \qquad (\beta\text{-riduzione})$$

$$\pi_1 \langle t_1, t_2 \rangle \to t_1 \qquad (\text{proiezione 1})$$

$$\pi_2 \langle t_1, t_2 \rangle \to t_2 \qquad (\text{proiezione 2})$$

Una dichiarazione di tipo è un'espressione del tipo $t : T$, che indica che il termine t ha tipo T. Sia Γ un insieme di dichiarazioni di tipo, detto contesto. Dato un contesto Γ possiamo fare delle inferenze di tipo su un termine t, in accordo con le seguenti regole di inferenza:

$$\frac{t : T \in \Gamma}{\Gamma \vdash t : T}$$

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash t_2 : T_2}{\Gamma \vdash \langle t_1, t_2 \rangle : T_1 \times T_2}$$

$$\frac{\Gamma \vdash t : T_1 \times T_2}{\Gamma \vdash \pi_1 t : T_1}$$

$$\frac{\Gamma \vdash t : T_1 \times T_2}{\Gamma \vdash \pi_2 t : T_2}$$

$$\frac{\Gamma, x : V \vdash u : U}{\Gamma \vdash \lambda x.u : V \to U}$$

$$\frac{\Gamma \vdash v : V \qquad \Gamma \vdash t : V \to U}{\Gamma \vdash tv : U}$$

Questo semplice esempio di lambda calcolo tipato è isomorfo al frammento $(\wedge, \Rightarrow)$ della Deduzione Naturale per la Logica Proposizionale Intuizionista. Le regole di eliminazione e di introduzione per il tipo coppia corrispondono alle regole di eliminazione e di introduzione del connettivo logico $\wedge$, e, analogamente, le regole di introduzione ed eliminazione del costruttore freccia corrispondono alle corrispettive regole per il connettivo logico $\Rightarrow$.

In questa analogia tra lambda calcolo tipato e calcolo logico, ai tipi corrispondono le formule ben formate della Logica Proposizionale, e ad una prova corretta corrisponde un termine ben tipato del calcolo. Dimostrare una proposizione corrisponde a trovare un termine del calcolo con il "tipo" della prova.

È possibile costruire degli alberi di inferenza di tipo, che permettono di stabilire se un termine t ha tipo T date una serie di assunzioni di tipo Γ. Questi alberi corrispondono ad alberi di derivazione in Deduzione Naturale di una proposizione T le cui ipotesi (i.e. foglie non cancellate) appartengono all'insieme dei tipi delle assunzioni contenute in Γ. Dato un albero di inferenza di tipo, ignorando i termini e considerando solo i tipi otteniamo una dimostrazione in Deduzione naturale.

Ad esempio, supponiamo di voler dimostrare che il termine $\lambda x : A.\lambda y : B.\langle x, y\rangle$ ha tipo $A \to B \to A \times B$ nel contesto vuoto. Questo può essere dimostrato con il seguente albero:

$$\cfrac{\cfrac{\cfrac{x : A, y : B \vdash x : A \qquad x : A, y : B \vdash y : B}{x : A, y : B \vdash \langle x, y\rangle : A \times B}}{x : A \vdash \lambda y : B.\langle x, y\rangle : A \times B : B \to A \times B}}{\vdash \lambda x : A.\lambda y : B.\langle x, y\rangle : A \times B : A \to B \to A \times B}$$

Ignorando i termini e, associando a $\times$ il connettivo $\wedge$ e a $\to$ il connettivo $\Rightarrow$, otteniamo una prova in Deduzione Naturale intuizionista di $A \Rightarrow B \Rightarrow A \wedge B$.

Il problema di verifica della correttezza di una prova corrisponde al problema del *type checking* di un termine del lambda calcolo, riassumibile nella domanda "dato un termine t ed un tipo T, t ha tipo T?", ed è tipicamente indicato con $t : T$?. Altri

problemi interessanti, legati ai lambda calcoli tipati, sono il problema della *tipability* di un termine ed il problema dell'*inhabitation* di un tipo, indicati, rispettivamente, da $t :?$ e $? : T$. Il primo è riassumibile dalla domanda "dato un termine t, esiste un tipo T tale che $t : T$?", mentre il secondo è riassumibile dalla domanda "dato un tipo T, esiste un termine t che ha tipo T?". Il secondo problema è particolarmente interessante, dato che corrisponde alla ricerca di una prova dato l'enunciato di un teorema. Per alcuni lambda calcoli tipati questi problemi sono decidibili, per altri possono non esserlo.

Questo semplice esempio rende l'idea di come un formalismo di calcolo possa essere utilizzato per verificare la correttezza di una prova in un calcolo logico. Alla base degli ITP sopra menzionati vi sono naturalmente dei lambda calcoli tipati molto più sofisticati.

1.2 Feature di Matita

Un ITP offre di base un supporto per la gestione delle prove, ma in generale non si limita a quello. I vari ITP sono diversi, ed offrono varie feature che dipendono dalla specifica implementazione. Nel caso di Matita tra le varie feature offerte vi è la rappresentazione grafica delle prove, in un'interfaccia grafica con tabs per i vari rami di prova ancora aperti. Matita offre inoltre una libreria standard contenente teoremi e definizioni per la formalizzazione di alcuni campi della matematica, quali la topologia e la teoria della calcolabilità basata su macchine di Turing.

L'interazione con Matita avviene mediante un'interfaccia testuale: l'utente può fornire, o scrivere da dentro Matita, uno script che contiene definizioni di tipo, di funzioni, di notazione, e di teoremi. Matita parserà lo script fornito, ed eseguirà i comandi specificati. Di particolare interesse ai fini di questo lavoro sono i comandi che permettono di costruire una dimostrazione.

Matita 0.5.x offre due forme di interazione agli utenti per la dimostrazione di teoremi: una è mediante il cosiddetto linguaggio procedurale, e l'altra è mediante il cosiddetto linguaggio dichiarativo.

Una prova viene iniziata dichiarando lo statement del teorema, che corrisponde ad un tipo in Matita. Questo tipo può essere costruito a partire da costruttori di tipo builtin (e.g. *dependent product*) o definiti dall'utente (e.g. O e $S(n)$ per la definizione

dei numeri naturali). Matita creerà, e gestirà, una metavariabile del tipo del teorema, che potremo andare a manipolare per costruire la nostra prova. La maniera in cui viene manipolata una metavariabile è l'applicazione di tattiche. Una tattica è, in termini informali, una funzione da un termine incompleto ad un altro termine incompleto, che ci permette quindi di passare da una prova incompleta ad una prova meno incompleta, con l'obiettivo di ottenere, alla fine della prova, un termine di prova completo. Ogni tattica, per essere applicata, ha dei prerequisiti, ed una volta applicata cambia il termine di prova da dimostrare. Quando tutti i rami di prova sono conclusi, ovvero quando si è giunti al termine della prova, il termine viene memorizzato in formato interno e salvato nella libreria del software per futuri utilizzi. In Matita esistono anche i tatticali, che possiamo considerare come delle tattiche di ordine superiore, ovvero che prendono in input tattiche o restituiscono nuove tattiche.

I due linguaggi di Matita corrispondono alle due famiglie di tattiche offerte dal software. Una prima famiglia di tattiche, che possiamo denominare procedurale, opera ad un "basso livello", offrendo delle manipolazioni precise e molto sofisticate, oltre a meccanismi di iterazione delle tattiche. In questo modo, è possibile, ad esempio, automatizzare una prova, aprendo in maniera programmatica tutti i rami di prova di una dimostrazione per casi che prevede centinaia di casi e applicare una serie di tattiche in maniera uniforme tra i casi. Questo genere di casi può presentarsi nell'utilizzo per la dimostrazione formale di correttezza del software, come, ad esempio, nel caso della dimostrazione della correttezza di un assemblatore C, con centinaia di istruzioni assembler.

Da un punto di vista sintattico, sono espresse con comandi più brevi, e questo permette di creare delle prove in maniera molto rapida e poco pedante. Vengono principalmente utilizzate nell'impiego di Matita in ambiti professionali, dove i dettagli della prova o il fatto che la prova sia leggibile ad un pubblico non specializzato non sono aspetti importanti.

Le tattiche dichiarative sono invece molte di meno, ma di grana più grossa rispetto a quelle procedurali. Corrispondono, circa, ai più comuni passi di inferenza di una tipica dimostrazione formale, quali, ad esempio, l'assunzione di ipotesi, i tagli logici, la giustificazione di passi di dimostrazione e la dimostrazione per induzione. Da un punto di vista sintattico sono più verbose, e sono espresse in un linguaggio molto vicino al

linguaggio naturale. Si prestano quindi all'utilizzo per la didattica, e all'utilizzo da parte di un pubblico più ampio, più familiare con il linguaggio naturale delle prove matematiche che con la sintassi dei comandi di Matita. Una prova dichiarativa, inoltre, risulta più semplice da leggere senza essere eseguita di una corrispondente prova procedurale, e questo risulta interessante ai fini della manutenibilità e dello studio di una prova già esistente.

1.3 Utilizzo nella didattica della Logica Matematica

All'università di Bologna Matita viene utilizzato per l'insegnamento della Logica Matematica durante il primo anno del corso di laurea triennale in Informatica. Matita permette agli studenti, mediante il linguaggio dichiarativo, di esercitarsi in semplici script di prova, che permettono di fare pratica con le più comuni tecniche di dimostrazione di teoremi utilizzate in vari contesti della Matematica e non solo.

Tramite Matita lo studente va a definire dei tipi di dato ricorsivi su cui andare poi a condurre delle dimostrazioni. Ad esempio, un esercizio prevede la definizione ricorsiva del tipo dei numeri naturali, per poi andare a definire la funzione somma mediante pattern matching sul tipo dei naturali, con lo scopo di dimostrare la commutatività della somma. Un altro esempio di esercizio, più interessante, prevede la definizione del tipo per le liste di numeri naturali, per poi andare a dimostrare la correttezza dell'algoritmo *Insertion Sort*.

Personalmente, ho trovato, da studente del corso di Logica, di grande utilità un Interactive Theorem Prover con cui esercitarmi in autonomia. Uno strumento come Matita può essere visto come una sorta di "compilatore" per teoremi e definizioni ricorsive, e, come esercitarsi nella programmazione con un compilatore è senz'altro più efficace che limitarsi allo studio delle tecniche di programmazione, trovo che esercitarsi sulle nozioni della Logica mediante uno strumento software del genere fornisca un valore aggiunto alla pratica della materia durante lo studio.

Naturalmente in questo utilizzo di Matita risulta di fondamentale importanza il linguaggio dichiarativo. Non ci aspetta che uno studente del primo anno di Informatica sia familiare con l'isomorfismo di Curry-Howard e con l'interpretazione dei lambda cal-

coli tipati come calcoli logici. Tuttavia, il linguaggio dichiarativo fornisce un'astrazione che maschera questi aspetti su cui si basa l'Interactive Theorem Prover, mediante una sintassi *English-like* che lo rende più accessibile.

Una delle ragioni che hanno portato a questo lavoro è quella di ripristinare e migliorare questo aspetto di Matita per la versione 0.99.x, in modo che gli studenti possano godere di tutti i benefici della nuova versione di Matita.

Capitolo 2

Architettura di Matita

2.1 Idea generale

L'idea generale, su cui si basa Matita, è quella di gestire termini di prova "incompleti", ovvero con delle parti non specificate, che andranno modificate per andare a creare un termine di prova completo. I "buchi" all'interno di una prova sono rappresentati da metavariabili all'interno di un termine incompleto, il cui valore può essere modificato mediante l'applicazione di tattiche. La verifica di correttezza di un termine completo viene fatta in una parte di codice che costituisce il Kernel di Matita, mentre la gestione dei termini incompleti viene agevolata mediante ciò che costituisce il refiner di Matita.

2.2 Componenti

Matita è composto da una serie di componenti che si suddividono il lavoro e comunicano tra loro. Tra le più importanti, e interessanti ai fini di questo lavoro, abbiamo il Kernel, l'Elaborator, il Refiner, le Tattiche ed il Parsing/Pretty Printing.

2.2.1 Kernel

Il Kernel rappresenta la trusted code base di Matita. È la parte di codice che implementa il calcolo logico su cui Matita è basato, il Calcolo delle Costruzioni (Co)Induttive,

e tutte le operazioni ad esso legato. In particolare, gestisce la riduzione di termini, la conversione, ed il type checking.

Assieme al kernel abbiamo la library, che si occupa di gestire la libreria di teoremi e definizioni, verificandone la correttezza prima di farne uso. Matita possiede una standard library, contenente definizioni e formalizzazioni di alcune teorie della Matematica.

2.2.2 Elaborator/refiner

L'elaborator/refiner si occupa di "aggiustare" termini abbozzati, ottenendo dei termini CIC corretti. Occorre qui fare una distinzione tra termini notazionali, e termini del nucleo. Mentre questi ultimi rappresentano l'implementazione, in OCaml, del calcolo logico su cui si basa Matita, e fanno parte del nucleo, i termini notazionali sono utilizzati per rappresentare i termini specificati dall'utente in un AST generato dal parser. Le prove sono sempre costituite da termini CIC, ma, durante la costruzione della prova, è possibile fare uso dei termini notazionali, che richiedono meno lavoro per essere definiti e utilizzati, grazie alle funzionalità offerte dal refiner.

Matita, mediante la sua interfaccia testuale, permette all'utente di specificare dei termini che non contengono tutti i dettagli necessari per costruire un termine di CIC. Ad esempio, un utente può specificare un termine senza tipo. Questo tipo verrà poi inferto dal contesto e dall'uso fatto del termine. Questa è una delle operazioni svolte dal refiner. Non è neanche necessario specificare tutti gli argomenti richiesti da un comando, lasciando che il refiner si occupi di riempire i buchi lasciati dall'utente. È possibile, ad esempio, utilizzare un carattere "?", che sta ad indicare un argomento non specificato, oppure tre puntini, che stanno ad indicare 0 o più argomenti non specificati. Ovviamente, il lavoro del refiner, che si basa su una serie di euristiche, non è banale, e può fallire anche in presenza di termini abbozzati che potrebbero, in linea di principio, essere aggiustati per ottenere un termine CIC.

Questa parte di codice si occupa della gestione di termini parziali, e delle metavariabili da utilizzare come placeholder per le parti di termine non specificate. Si occupa anche di unificazione tra termini, ovvero, dati due termini parziali t_1 e t_2 ed un'equazione $t_1 = t_2$, cerca di trovare una soluzione, istanziando le metavariabili in t_1 e t_2 in modo tale che i due termini diventino convertibili. Si tratta di un problema indecidibile, che viene

affrontato mediante una serie di euristiche. Il refiner, dato un termine abbozzato, cerca di completarlo in modo da ottenere un termine CIC, andando a riempire l'informazione omessa, e facendo uso, ove possibile, dell'unificazione.

Questa parte di codice permette all'utente di evitare di essere pedante nella specifica di un termine, e permette di implementare un linguaggio per l'interazione utente ad un livello più alto rispetto a quello del calcolo logico su cui il sistema si basa.

2.2.3 Tattiche

Le tattiche rappresentano lo strumento fondamentale di interazione dell'utente con le prove. Si possono vedere come dei comandi ad alto livello (rispetto al calcolo logico sottostante) mediante i quali l'utente va a riempire i placeholder all'interno di un termine incompleto per andare a costruire una prova completa. Il termine parziale che viene a costruirsi viene dato in input al refiner affinché venga aggiunta l'informazione eventualmente omessa.

Abbiamo fondamentalmente due tipi di tattiche:

- tattiche procedurali, di "basso livello", poco verbose, molto precise in quello che fanno, adatte ad un utilizzo professionale di Matita;
- tattiche dichiarative, di "alto livello", verbose, più vicine al linguaggio naturale, adatte ad un utilizzo didattico o per leggere una prova già formata in un linguaggio più alto di quello delle tattiche procedurali applicate per costruire la prova.

Assieme alle tattiche in questa componente viene gestita la struttura dei goal aperti, ovvero dei rami della prova in corso non ancora non completati da parte dell'utente. Mentre le parti più basse dell'architettura (i.e. Kernel, Elaborator, Refiner) lavorano su termini CIC e termini dell'AST del parser di Matita, questa parte del sistema software lavora sui goal, e sullo "status" della prova in corso, che contiene l'insieme di tutte le informazioni interessanti legate ad una prova (tra cui istanziazione di metavariabili, goal aperti/chiusi, focus dei goal, etc.).

Un'altra componente legata a questa parte del sistema software è l'automazione di Matita, basata su una tecnica denominata paramodulazione. Fondamentalmente ciò che

fa la paramodulazione è, dato un insieme di uguaglianze tra termini, cerca di costruire un sistema di riscrittura convergente, mediante il quale dimostrare uguaglianze del tipo $E_1 = E_2$, sulla base di assiomi $F_1 = F_2, \ldots$. Costruire un sistema convergente per un dato insieme di equazioni non è sempre possibile, ma si può approssimare un sistema convergente mediante una tecnica denominata *ordered completion*. Matita si basa un algoritmo denominato *given-clause algorithm* [7] per risolvere questo problema, che rimane generalmente indecidibile.

L'automazione, in Matita, ha due scopi principali:

1. andare a riempire i dettagli "triviali" di una prova; si parla in questo caso di *small scale automation*;

2. lanciare la ricerca di una prova in maniera automatica; si parla in questo caso di *large scale automation*.

Per i fini didattici e illustrativi del linguaggio dichiarativo, la funzione più interessante è senz'altro la prima.

2.2.4 Parsing/Pretty printing

Questa parte di Matita si occupa del parsing dei comandi dell'utente e del pretty printing di termini. Matita implementa un parser $LL(k)$, con k variabile, basato su camlp5, in modo da avere un parser leggero ed espandibile. Il lookahead non è costante e stabilito a priori, ma variabile, in base alla necessità. Uno svantaggio legato a questo parser è che parsing e pretty printing sono aspetti separati, e vanno implementati uno indipendentemente dall'altro, senza avere, a priori, la garanzia che il risultato del pretty printing sia accettabile dal parser.

Una feature offerta da Matita mediante questa componente è la disambiguazione dell'input dell'utente e la gestione dell'overloading della notazione matematica.

Un'altra parte interessante del sistema Matita è la generazione di una prova dichiarativa a partire da un termine CIC. In questo modo è possibile rendere, in maniera più "leggibile", un termine di prova già costruito. Questo rende possibile, ad esempio, lo scambio di termini con un theorem prover basato sullo stesso calcolo logico (e.g. Coq)

per la resa dichiarativa di una prova, oppure permette di costruire una prova mediante automazione, per poi renderla in linguaggio dichiarativo per studiarla e, eventualmente, modificarla.

Tutto ciò che si occupa dell'input utente in matita va sotto il nome di Grafite. Esistono tre "moduli" di Grafite:

1. Grafite AST, che definisce i tipi per l'albero di sintassi astratta costruito dal parser, compresi i tipi per le tattiche;

2. Grafite Parser, che costituisce il parser vero e proprio;

3. Grafite Engine, che contiene i "binding" tra parsing di un'espressione ed esecuzione di codice del core di Matita.

Un'ulteriore feature interessante legata a questa componente software è l'estrazione di codice a partire da una prova. Dal punto di vista dell'isomorfismo di Curry-Howard, ad una dimostrazione intuizionista di correttezza di una specifica di un programma corrisponde un'implementazione della specifica. Questa implementazione può essere generata automaticamente in Matita nel linguaggio OCaml o Haskell. A dimostrazioni di correttezza diverse corrispondono algoritmi diversi: ad esempio, ad una dimostrazione di correttezza di un algoritmo di sorting su liste può corrispondere il bubble sort, mentre ad un altra può corrispondere il merge sort.

2.3 Differenze tra Matita 0.5.x e Matita 0.99.x

Nel passaggio dalla versione 0.5.x alla versione 0.99.x vi sono stati vari cambiamenti strutturali.

Un cambiamento importante è stato la semplificazione dei tipi di CIC nel kernel. In particolare, in Matita 0.5.x vi era una "doppia definizione" dei tipi dei termini di CIC, con un tipo che rappresentava un termine, ed un tipo analogo che rappresentava un termine annotato. I termini annotati contenevano un'informazione ulteriore riguardo al tipo che permetteva, in fase di ricostruzione di una prova dichiarativa a partire da un termine, di risolvere situazioni di ambiguità nella ricostruzione della prova. Questa

feature è stata rimossa in Matita 0.99.x, con il vantaggio di una semplificazione del codice del Kernel, ma con lo svantaggio di aver reso più complessa la ricostruzione di prove a partire da termini CIC completi.

Un altro cambiamento interessante ai fini di questo lavoro è legato all'implementazione delle tattiche. Mentre in Matita 0.5.x tattiche come la **exact_tac**, che permette di istanziare direttamente una metavariabile, richiedevano tipi CIC completi come argomento, in Matita 0.99.x si è deciso di optare per una flessibilità maggiore, permettendo la specifica di termini notazionali come argomenti. Questi termini notazionali vengono poi elaborati dal refiner prima di entrar a far parte del termine. Questo "cambio di paradigma" ha avuto una serie di conseguenze positive, che saranno esplorate nel Capitolo 3.

Capitolo 3

Implementazione

In generale l'implementazione del linguaggio dichiarativo è consistita nelle seguenti attività per ognuna delle tattiche dichiarative descritte in [10]:

- aggiunta di un tipo nel Grafite AST per la tattica; durante il parsing di uno script, gli oggetti parsati vengono gestiti nel codice mediante una rappresentazione interna che permette di fare uso del pattern matching di OCaml per eseguire il codice adeguato;
- aggiunta del codice di pretty printing, che, dato un AST, restituisce la sua versione testuale, riparsabile nell'AST di partenza;
- aggiunta delle regole alla grammatica del Grafite Parser per il parsing della tattica;
- aggiunta delle keyword della tattica alle keyword da evidenziare in Matita mediante syntax highlighting; l'interfaccia grafica di Matita è basata su GTK, che permette di fare un syntax highlighting basato su string matching;
- aggiunta della documentazione per la tattica nell'help online di Matita;
- aggiunta di chiamate in Grafite Engine all'implementazione della tattica, affinché la tattica venga effettivamente eseguita durante la compilazione dello script;
- implementazione vera e propria della tattica, che, a partire dallo status al momento in cui è stata raggiunta la tattica nello script, restituisce un nuovo status modificato in base alla semantica della tattica.

Di queste attività quella sicuramente più interessante e che ha richiesto, in generale, maggior lavoro, è senz'altro l'implementazione vera e propria della tattica, e infatti sarà oggetto delle prossime sezioni. Di base, partendo dall'implementazione in Matita 0.5.x, è stato fatto un lavoro di ripristino della tattica, con l'obiettivo di sistemare alcuni aspetti scorretti della vecchia implementazione e di realizzare l'implementazione nella maniera più naturale e pulita possibile, per futura manutenibilità e leggibilità del codice.

3.1 Due approcci per implementare una tattica

L'effetto di una tattica può essere la modifica del contesto del goal aperto corrente (e.g. introduzione di ipotesi), la modifica della conclusione del goal corrente (e.g. assegnamento di un termine alla metavariabile del goal), o una modifica dello status di altra natura (e.g. branching per cambiare il goal sotto focus). Per implementare la semantica di una tattica è possibile agire in due modi:

1. utilizzando altre tattiche, ad esempio procedurali, il cui effetto combinato corrisponde alla semantica della tattica;

2. istanziando direttamente la metavariabile del goal aperto correntemente con un lambda termine costruito al momento dell'esecuzione della tattica.

Mentre l'approccio 1 può essere applicato in maniera quasi identica in Matita 0.99.x, a meno di ridenominazione delle tattiche usate, l'approccio 2 viene applicato in maniera diversa tra le due versioni di Matita. In Matita 0.5.x, il termine t con cui si vuole istanziare la variabile deve essere un termine CIC, completamente specificato in tutti i suoi dettagli. Questo richiede del lavoro non banale per raccogliere le informazioni di tipo adeguate necessarie a costruire il termine, e a questo approccio è legato il problema del lifting.

Il lifting è un processo che consiste nell'estrarre un lambda sottotermine t_1 da un termine t, in modo che il termine estratto t'_1 sia indipendente da t e che si comporti come t_1. Questo processo è necessario quando si vuole utilizzare un termine t_1, con contesto Γ_1, in un nuovo contesto Γ_2, poiché, ad esempio, t_1 in Γ_2 non può dipendere da variabili in Γ_1. Questo processo richiede, solitamente, di eliminare le variabili libere in t_1,

aggiungere delle opportune lambda astrazioni, ottenendo un t'_1 "globale" che corrisponde al t_1 "locale", per poi calare t'_1 in Γ_2 istanziando opportunamente le variabili in t'_1 che corrispondevano a variabili libere in t_1. Se t_1 contiene delle variabili presenti in Γ_1, e Γ_2 contiene più variabili rispetto a Γ_1, è possibile calare t_1 in Γ_2. In caso Γ_2 contenga meno variabili di Γ_1, abbiamo una situazione di errore da gestire in maniera appropriata.

In Matita 0.5.x questo processo veniva fatto manualmente quando necessario. In Matita 0.99.x è stata introdotta una nuova struttura dati, che consiste di una coppia formata da un lambda termine ed un contesto, in modo da implementare una tecnica denominata "autolifting". Non si tratta di un problema banale, e la sua soluzione in Matita è basata su alcune euristiche.

Supponiamo di avere termine t_1 in un contesto Γ_1, e di volerlo copiare in un contesto Γ_2. t_1 non può rimanere identico, deve essere *lifted* da Γ_1 a Γ_2. Mentre in Matita 0.5.x sarebbe stato necessario effettuare il lifting manualmente nel codice, in Matita 0.99.x, quando un termine t viene utilizzato in un contesto Γ_2 diverso dal suo contesto Γ_1, vengono utilizzate delle euristiche per effettuare un auto-lifting da Γ_1 a Γ_2. Di conseguenza, utilizzare un termine in un nuovo contesto richiede meno lavoro e meno codice rispetto alla versione 0.5.x di Matita.

In Matita 0.99.x, inoltre, è possibile istanziare una metavariabile con un termine notazionale ottenuto dall'AST prodotto dal parsing o costruito al momento nel codice. In questo caso il refiner verrà utilizzato per "aggiustare" il termine incompleto, ed ottenere un termine CIC con cui costruire il termine di prova. Questo metodo di istanziazione dipende molto dall'"intelligenza" di Matita nel disambiguare l'input dell'utente e nel saper costruire termini completi in mancanza di alcune informazioni, che può essere dovuta a non specificazione del tipo, utilizzo di argomenti impliciti, overloading, etc. Di conseguenza, risulta più semplice l'istanziazione diretta di una metavariabile con un termine t, dato che può essere utilizzato un termine notazionale con dell'informazione mancante, piuttosto che un termine CIC completamente specificato.

L'approccio 2 di Matita 0.99.x è senz'altro più semplice e più flessibile, ma allo stesso tempo più fragile, in quanto è più soggetto ad errori dovuti ad un lambda termine errato fornito dall'utente.

L'approccio 2 di Matita 0.5.x ha il vantaggio di una maggiore precisione, ma richiede

lifting, e rende le tattiche, in un certo senso, più "stupide", in quanto, man mano che il refiner viene migliorato per comprendere maggiormente l'input dell'utente e raffinarlo in maniera più accurata, le tattiche non rimangono al passo con l'aggiornamento del sistema, utilizzando meccanismi diversi e "hard coded" nell'implementazione delle tattiche per l'istanziazione di metavariabili.

In Matita 0.99.x i due approcci per l'istanziazione di lambda termini non sono mutualmente esclusivi: possono coesistere senza problemi. Nel codice di Matita 0.99.x vi è tuttavia la tendenza a preferire l'approccio basato su termini notazionali per l'implementazione delle tattiche.

3.2 Tattiche LCF e tinycals

Uno dei più influenti Interactive Theorem Prover basati su tattiche è senz'altro stato Logic for Computable Functions (LCF) [2]. LCF è stato uno dei primi ITP ad introdurre il concetto di *tatticale*, o tattica di ordine superiore. Molti ITP basati su tattiche si sono ispirati a LCF per l'implementazione di tattiche e tatticali. Matita si ispira anch'esso a LCF, ma sono state apportate delle modifiche alle implementazioni di tattiche e tatticali per superare alcune mancanze delle implementazioni originali di LCF.

Uno degli svantaggi dell'implementazione di LCF dei tatticali è che l'applicazione dei tatticali era atomica, rendendo molto faticoso andare ad ispezionare gli effetti singoli delle tattiche argomento di un tatticale. Ad esempio, nell'applicare il tatticale di *sequential composition*, che, date due tattiche t_1 e t_2, applica t_1 al goal corrente e applica poi t_2 a tutti i goal aperti dall'applicazione di t_1, era possibile solo vedere l'effetto totale del tatticale, e non era possibile andare a vedere l'effetto di t_2 su uno dei goal aperti da t_1.

Per risolvere questo problema, e per una serie di altri vantaggi, Matita è basato sui *tinycals* [11], o *step-by-step tacticals*. I tinycals "spezzano" un tatticale, permettendo di andare ad agire sugli stati intermedi prodotti dall'applicazione del tatticale. Il funzionamento dei tinycals è basato su una struttura dati detta *stack*.

Altri punti deboli dell'implementazione originale delle tattiche LCF sono descritti in [6]. Ai fini di questo lavoro, di particolare interesse è il fatto che le tattiche LCF agiscono su un solo goal alla volta, e sono completamente locali. Questo significa che, ad esempio,

se esistono due goal aperti che condividono una metavariabile, per le tattiche LCF questi due goal sono completamente indipendenti. Questo rappresenta un problema, poiché, in generale, i due goal non sono indipendenti, e, per trovare una prova, è necessario istanziare le metavariabili dei goal in virtù di queste dipendenze.

Ad esempio, siano t_1 e t_2 due termini e sia $\leq$ un preordine per il tipo, e supponiamo di voler dimostrare $t_1 \leq t_2$ usando la transitività della relazione $\leq$ e andando a dimostrare che $t_1 \leq ?1$ e $?1 \leq t_2$. Dal punto di vista delle tattiche LCF, potremmo istanziare ?1 con t_1 nel primo goal, e trovarci a dimostrare nuovamente $t_1 \leq t_2$ nel secondo goal. Ma questo passaggio non ha, ovviamente, alcuna utilità. Questo genere di situazioni sono particolarmente rilevanti nell'impiego di un motore di automazione nella ricerca delle prove.

In Matita le tattiche sono, di default, globali, e possono andare ad agire su più di un goal alla volta. Questo ha sicuramente dei vantaggi nel caso procedurale, quali una migliore automazione e una maggiore concisione nella scrittura di prove, ma risulta svantaggioso nel caso del linguaggio dichiarativo. Infatti, in una prova dichiarativa possono essere aperti dei goal durante la prova, e ognuno dei goal aperti viene dimostrato in autonomia rispetto agli altri. Di conseguenza, per il linguaggio dichiarativo, la semantica delle tattiche LCF, che opera su un goal alla volta, risulta corretta.

Per implementare le tattiche dichiarative è stato sfruttato il meccanismo dei tinycals per implementare delle tattiche LCF-like, che agiscono su un solo goal alla volta. Questa implementazione si basa sui tinycals *branch* e *merge*, che permettono di cambiare i goal sotto focus e chiudere rami di prova in una maniera "strutturata".

3.3 Stack dei *tinycals*

In Matita per la gestione dei goal aperti viene utilizzata una struttura dati denominata *stack*. Questa struttura dati è stata introdotta inizialmente in [11] per l'implementazione dei *tinycals*, ed è implementata mediante una lista di *entry*. Ogni *entry*, o livello, contiene 3 liste di *locator* ed un *tag*. Un locator consiste di una coppia formata da un intero ed uno *switch*. Uno switch è un tipo di dato algebrico con due costruttori: *Open* e

Closed, per indicare lo stato del goal. Entrambi questi due costruttori richiedono, come argomento, un intero che rappresenta un goal.

Le tre liste di goal hanno un significato specifico a loro associato: la prima lista, che indicheremo con Γ, rappresenta la lista di goal sotto focus al momento; la seconda lista, che indicheremo con τ, rappresenta la lista di goal "lasciati indietro" in un contesto di branching, e la terza lista, che indicheremo con κ, è utilizzata per l'implementazione della tattica LCF *dot*.

Quando si comincia una prova viene creato uno stack con un solo livello, con il solo goal della prova nella lista Γ, con le locator list τ e κ vuote, e con il tag `NoTag`. Matita pone sempre il focus su tutti i goal nella lista Γ dell'ultimo livello dello stack. Quando una tattica t_1 apre una serie di nuovi goal, questi vengono tutti inseriti in Γ.

Per andare ad eseguire delle nuove tattiche sui goal aperti da t_1, si fa tipicamente uso, in ITP ispirati a LCF, di tatticali. Di particolare interesse, tra i tatticali LCF, sono i tatticali di *branching* e *sequential composition*. Il primo prende come argomenti $n + 1$ tattiche, indicate con $t, t_1, \ldots, t_n$, e va ad eseguire la tattica t, aspettandosi che vengano aperti n nuovi goal. Ad ogni nuovo goal i aperto viene applicata la tattica t_i. Il secondo prende in input due tattiche t_1 e t_2, applica t_1, ed applica poi t_2 ad ognuno dei goal aperti da t_1. Mentre in LCF questi sarebbero eseguiti atomicamente, e richiederebbero come argomento tutte le tattiche da applicare, in Matita è possibile "destrutturare" la loro applicazione.

Se vi è più di un goal sotto focus, è possibile applicare il tinycal *branch*. Quando questo viene eseguito viene aggiunto un nuovo livello allo stack, con tag `BranchTag`, ed il primo dei goal nella lista Γ del livello subito inferiore viene pushato nella lista Γ del nuovo livello. Questi due livelli consecutivi formano un contesto di branching.

È possibile a questo punto applicare nuove tattiche sul goal sotto focus, fino a chiuderlo. Per passare al prossimo goal nella lista si usa il tinycal *shift*, che prende un goal dalla lista Γ del livello precedente e lo pusha nella lista Γ del livello corrente. Il tinycal *merge* fa pop di un livello dello stack, andando ad inserire tutti i goal rimasti aperti nell'ultimo livello nella lista Γ del livello subito precedente.

Non è necessario andare a lavorare sui goal nell'ordine in cui sono stati aperti e applicando tattiche su un goal alla volta. In un contesto di branching si può utilizzare

il tinycal *pos* per porre il focus su uno o più goal. I goal sono indicizzati, in un contesto di branching, dall'intero contenuto nel locator del goal, ed è possibile riferirsi ai goal mediante i loro indici nel tinycal pos. Si può anche fare shifting pur non avendo chiuso i goal correntemente sotto focus. Questi goal rimasti aperti verranno spostati nella lista τ del livello corrente, e verranno reinseriti nella lista Γ del livello subito inferiore nello stack dopo l'applicazione del tinycal merge.

È possibile, in un momento qualsiasi del branching, porre il focus sull'insieme di goal formato da tutti i goal correntemente sotto focus e sui goal della lista Γ del livello precedente, mediante il tinycal *wildcard*. Questo è utile in uno scenario dove, dopo aver applicato una serie di tattiche specifiche in alcuni goal, si vuole lavorare sui goal rimanenti nel corrente contesto di branching in una maniera uniforme.

Esiste inoltre un'implementazione della tattica LCF *dot*, che permette di andare a lavorare sui goal uno alla volta ed in sequenza senza aprire un contesto di branching. Questa tattica, quando applicata per la prima volta, sposta tutti i goal nella lista Γ del livello corrente, ad eccezione del primo, nella lista κ. Applicazioni successive della tattica non hanno effetto se esiste un goal nella lista Γ, e, se la lista Γ è vuota e la lista κ contiene dei goal, la dot sposta il primo goal di κ in Γ.

In alternativa alla coppia di tinycals branch/merge, esiste la coppia di tinycals *focus*/*unfocus*, che permette di focalizzarsi su un goal diverso da quello corrente. Anche focus, quando applicato, aggiunge un nuovo livello allo stack, con tag `FocusTag`. Poiché entrambe le coppie di tattiche per la focalizzazione su goal sono disaccoppiate nel parsing, risulta necessario poter distinguere quando è stato applicato un branch o un focus, da cui la necessità del tag per ogni livello dello stack.

La differenza principale tra le due coppie di tinycals sta nel fatto che, mentre branch permette di focalizzarsi solamente sui goal della lista Γ dell'ultimo livello dello stack, focus permette di focalizzarsi su un goal aperto qualsiasi nella prova, e quindi a livelli dello stack diversi dall'ultimo. Mentre l'approccio basato su branch/merge rappresenta una maniera strutturata di costruire una prova, l'approccio basato su focus/unfocus risulta non strutturato, ma permette all'utente di svolgere una prova in un ordine completamente arbitrario. Il tatticale branch rispetta, in un certo senso, la struttura ad albero della prova, a differenza della focus.

Quando un goal viene focalizzato mediante focus, questo non viene spostato nell'ultimo livello dello stack, bensì viene copiato. Di conseguenza, quando il goal chiuso mediante focus verrà nuovamente raggiunto durante la prova, risulterà chiuso per via di un side effect, e dovrà essere accettato mediante il comando *skip*.

L'esecuzione dei tinycals è atomica. Mediante questa gestione dei goal si va a spezzare tatticali come *branching* e *sequential composition*, permettendo all'utente di vedere l'effetto dell'applicazione di tattiche ad uno o più dei goal aperti dall'applicazione di una tattica precedente. Questa destrutturazione ha vari effetti positivi sulla struttura degli script di prova e sulla loro esecuzione, come descritto in [11].

Lo stack rappresenta una delle due strutture dati che contengono informazioni interessanti legate ad una prova. L'altra struttura dati interessante è lo *status*. Gli effetti dell'esecuzione di tattiche e tinycals vanno a modificare queste due strutture dati nel corso della prova. Per questa implementazione del linguaggio dichiarativo la gestione dello stack è stata modificata, per venir incontro ad alcune problematiche che sorgono nella gestione di goal multipli e nella gestione di analisi per casi e dimostrazioni per induzione nel caso dichiarativo.

Sebbene i tinycals permettano di spezzare il flusso della dimostrazione ed offrano la possibilità di dimostrare i goal aperti nell'ordine scelto dall'utente, dal punto di vista del linguaggio dichiarativo questi aspetti non risultano particolarmente rilevanti. Infatti, nel linguaggio dichiarativo il flusso di dimostrazione è più lineare: in generale i goal vengono dimostrati nell'ordine in cui vengono aperti. Fanno eccezione a questo flusso lineare l'analisi per casi e per la dimostrazione per induzione, dove è possibile dimostrare i casi nell'ordine scelto dall'utente.

3.4 Novità introdotte

In questo lavoro di implementazione del linguaggio dichiarativo sono state apportate alcune modifiche allo stack e sono stati introdotti dei nuovi concetti. Queste novità hanno reso più agevole l'implementazione della semantica di alcune tattiche, e hanno permesso di portare alcune correzioni rispetto alla semantica implementata da Matita 0.5.x.

3.4.1 Parameter list per lo stack dei tinycals

Durante una prova accade spesso di voler associare delle informazioni arbitrarie alla prova corrente. Nonostante questa possibilità fosse prevista in [6], sotto forma di tag come variante polimorfa aperta per associare informazioni ad un livello dello stack dei tinycals, nell'implementazione di Matita 0.5.x questo aspetto non era effettivamente presente. Di conseguenza, l'implementazione dello stack è stata modificata per accomodare questa possibilità.

Un problema che sorge in questo contesto è legato al fatto che, sebbene si voglia associare un'informazione arbitraria ad un contesto di prova, nel codice OCaml è necessario specificare un tipo preciso. Di conseguenza è necessario trovare il miglior compromesso per permettere di associare informazioni al contesto di prova corrente, nelle limitazioni imposte dal sistema di tipi di OCaml. Nel tentativo di fornire una soluzione sufficientemente generale per questa problematica si è optato per una lista di coppie chiave-valore, composte da stringhe. Poiché il numero di chiavi associate ad una prova in generale è molto ristretto, non si ha un calo di performance apprezzabile per via dell'utilizzo di una lista, piuttosto che un dizionario basato su hash table o alberi bilanciati.

L'utilizzo primario della parameter list è, al momento, quello di memorizzare il "contesto" di una prova. In questa accezione del termine contesto non si intende l'insieme delle ipotesi, bensì un insieme di informazioni generali sullo stato corrente della prova che permettono di condizionare il comportamento di alcune tattiche. Data la generalità della struttura dati è possibile, se necessario, associare informazione arbitraria ad un livello dello stack, a patto che sia esprimibile sotto forma di stringa.

3.4.2 Nuova gestione dei livelli dello stack

Una problematica che è sorta con la vecchia implementazione delle tattiche dichiarative è legata alla gestione di goal multipli. Quando le tattiche dichiarative **we need to prove** (con nome di ipotesi per effettuare un taglio logico), **we proceed by cases/induction** e **obtain** vengono applicate, e se hanno successo, aprono più goal, ognuno dei quali deve essere dimostrato in autonomia rispetto agli altri. Le tattiche dichiarative, infatti, hanno la semantica di una tattica LCF, dove lo scope di applicazione della

tattica consiste di un solo goal. In Matita questo non è vero in generale, in quanto le tattiche procedurali e l'automazione lavorano su più di un goal contemporaneamente, poiché spesso i goal aperti non sono indipendenti tra loro, e devono essere dimostrati simultaneamente.

In Matita 0.5.x le tattiche dichiarative che aprivano più di un goal venivano gestite mediante la tattica dot. In questo modo i goal aperti venivano gestiti uno alla volta, ma la struttura risultante, dal punto di vista dello stack dei tinycals, risultava piatta: tutti i goal stavano in un solo livello dello stack. Per implementare una semantica più interessante per queste tattiche, che creano un livello di annidamento nella prova, sarebbe ideale che ad ognuna di queste tattiche corrispondesse un livello nello stack.

Un tale obbiettivo è raggiungibile facendo uso dei tinycals branch/merge. Per gestire i goal multipli mediante branch, e andare a lavorare sui goal in una maniera strutturata, è stata modificata la gestione dello stack durante l'applicazione di una delle tattiche dichiarative sopra citate. Quando viene aperto un nuovo contesto di branching, infatti, i goal risultano distribuiti tra le locator list di due livelli dello stack. Questo è problematico, dato che vorremmo che ad ognuna di queste tattiche corrispondesse un solo livello. Eventuali informazioni contestuali, ad esempio, andrebbero altrimenti distribuite tra le due parameter list dei due livelli dello stack.

Nella nuova implementazione viene fatto uso della seconda locator list dei livelli dello stack. In una dimostrazione dichiarativa, infatti, nessun goal viene mai lasciato indietro: quando viene posta l'attenzione su un goal questo deve essere chiuso prima di spostarsi su altri goal aperti. Di conseguenza, la seconda locator list risulta essere sempre vuota. Questa lista viene quindi utilizzata per mantenere i molteplici goal aperti dall'applicazione di una tattica dichiarativa.

Quando una tattica dichiarativa apre più di un nuovo goal, viene aggiunto un nuovo livello allo stack e tutti i nuovi goal aperti pushati nella lista τ del nuovo livello. Per passare al goal successivo del livello corrente, dopo, ad esempio, aver fatto un taglio logico con la tattica **we need to prove** e dopo aver dimostrato l'ipotesi di taglio, viene spostato un goal dalla lista τ alla lista Γ del livello corrente, in maniera analoga all'esecuzione della tattica LCF dot.

Un goal viene sempre chiuso dalla tattica **done**. Quando questa tattica viene ap-

plicata, è necessario, nell'implementazione, scegliere se passare a dimostrare un goal successivo, o se chiudere il contesto corrente di branching. Nel secondo caso, si torna indietro di un livello nella ramificazione della prova, e in questo livello è necessario fare nuovamente una scelta tra le due alternative. Per capire quale delle due strade prendere è sufficiente guardare la locator list τ: se questa è vuota, è necessario chiudere il contesto corrente con una merge; altrimenti, viene spostato un goal da τ a Γ nel livello corrente.

Se il livello di annidamento nella prova dichiarativa è n (e.g. è stata applicata n volte la **we proceed by cases** e gli ultimi casi dei livelli precedenti sono ancora aperti), e viene applicata la done sull'ultimo caso del livello di annidamento corrente, verranno chiusi, a cascata, tutti i livelli precedenti. Dopo aver applicato una done, ci si può ritrovare a dimostrare eventuali goal aperti dei livelli precedenti di annidamento, o si può giungere alla conclusione della prova.

Per le dimostrazioni per induzione/casi vi è una gestione particolare: per implementare la semantica desiderata per queste tattiche, in seguito alla chiusura di un caso il sistema non passa automaticamente al prossimo goal. Bensì, l'utente deve specificare esplicitamente il prossimo caso per proseguire.

3.4.3 Informazioni contestuali e contesti volatili

Una delle informazioni che vogliamo spesso associare ad una prova nel suo stato corrente è il contesto della prova in cui si trova. Infatti, in base alla tattica appena applicata, o al contesto di induzione o analisi per casi in cui si trova l'utente, potremmo voler permettere all'utente di cominciare una catena di beta riduzioni (i.e. semplificazioni), oppure di passare a dimostrare un caso dell'induzione. Vi sono inoltre alcune tattiche che possono essere eseguite solo in particolari contesti, e vorremmo poter impedire all'utente di utilizzarle al di fuori di essi. Per fare ciò viene sfruttata la parameter list, utilizzando il parametro con chiave *context*. Questo parametro può assumere i seguenti valori:

- *beta_rewrite* e sta ad indicare che ci si trova nel contesto di una catena di beta riduzioni; con questo contesto è possibile utilizzare la tattica **that is equivalent to**, per modificare la conclusione corrente;

- *rewrite* e sta ad indicare che ci si trova nel contesto di una catena di uguaglianze; con questo contesto è possibile utilizzare la tattica di uguaglianza per proseguire nella catena;

- *induction* o *cases* e sta ad indicare che ci si trova nel contesto di una dimostrazione per induzione, o per casi; con questo contesto è possibile utilizzare la tattica *case* per focalizzarsi su un caso della dimostrazione.

Un altro parametro che fornisce informazioni contestuali ha chiave *new_hypo*, che sta ad indicare che è appena stata introdotta un'ipotesi nel contesto. Il valore del parametro è il nome dell'ipotesi. Con questo parametro è possibile condurre passi di beta riduzione sull'ipotesi introdotta. Questa catena viene interrotta una volta applicata una tattica che la "spezza". In questo caso viene sfruttato il meccanismo dei parametri volatili.

3.4.4 Parametri volatili

Molto spesso capita che il flusso di una dimostrazione sia spezzato dall'applicazione di una tattica e si passi a dimostrare altro. In questi casi eventuali catene di uguaglianza o beta riduzione risulterebbero innaturali se continuassero dopo che il flusso della dimostrazione è stato interrotto. Ad esempio, risulterebbe innaturale usare la tattica **we need to prove** per cambiare la conclusione, usare poi la tattica **we proceed by cases** per continuare la dimostrazione, e proseguire con la catena di beta riduzioni della conclusione con **that is equivalent to**.

Le tattiche di Matita, tuttavia, non hanno uno stato a loro associato: vengono applicate indipendentemente da quali tattiche sono state applicate prima, e non hanno effetti che modificano il comportamento di tattiche applicate successivamente. Per fare in modo che l'applicazione di una tattica dipenda dalle tattiche applicate in precedenza viene fatto uso della parameter list. In particolare, in questo caso viene fatto uso del meccanismo del parametri volatili.

Alcuni parametri nella parameter list possono essere contrassegnati come "volatili", prefiggendoli con "volatile_" nella chiave. Questo ha l'effetto di rendere il parametro soggetto ad un eventuale clear se viene applicata una tattica che spezza il flusso della

dimostrazione. Infatti, quando viene applicata una tattica che apre un nuovo contesto, o ne spezza uno esistente, i parametri volatili vengono eliminati dalla parameter list.

Esempi di utilizzo di un contesto volatile sono le tattiche che permettono di fare beta riduzione su un'ipotesi o sulla conclusione, oppure le catene di uguaglianze.

3.5 Assume e Suppose

La tattica assume permette di aggiungere al contesto della prova corrente una nuova ipotesi $x : T$, se la conclusione è una quantificazione universale su x con tipo T. La sintassi è la seguente:

assume $x : T$

La tattica suppose permette di aggiungere al contesto della prova corrente una nuova ipotesi A con un identificatore H se la conclusione è un'implicazione logica di cui A rappresenta la premessa. La sintassi è la seguente:

suppose A (H)

Dal punto di vista di CIC, gli effetti di queste due tattiche sono simili, in quanto sia l'implicazione che la quantificazione universale corrispondono ad un prodotto in CIC. CIC è un calcolo che si trova "in cima" al lambda cube di Barendregt [9]: di conseguenza è possibile astrarre sia su tipi che su termini.

In Matita 0.5.x queste due tattiche erano implementate, facendo riferimento agli approcci descritti in 3.1, mediante l'approccio 1. Il motivo principale dietro a ciò è che in Matita vecchio per andare a sostituire una metavariabile con un lambda termine era necessario specificare completamente un termine di CIC, senza tralasciare dettagli e richiedendo un lavoro non banale.

In Matita 0.99.x, sfruttando il nuovo approccio implementativo, che permette di specificare un termine in maniera incompleta, lasciando poi al sistema il compito di trasformare questo termine in un termine CIC completamente specificato, è possibile utilizzare l'approccio 2, andando a sostituire alla conclusione un termine notazionale che rappresenti una lambda astrazione. Questo è stato l'approccio seguito.

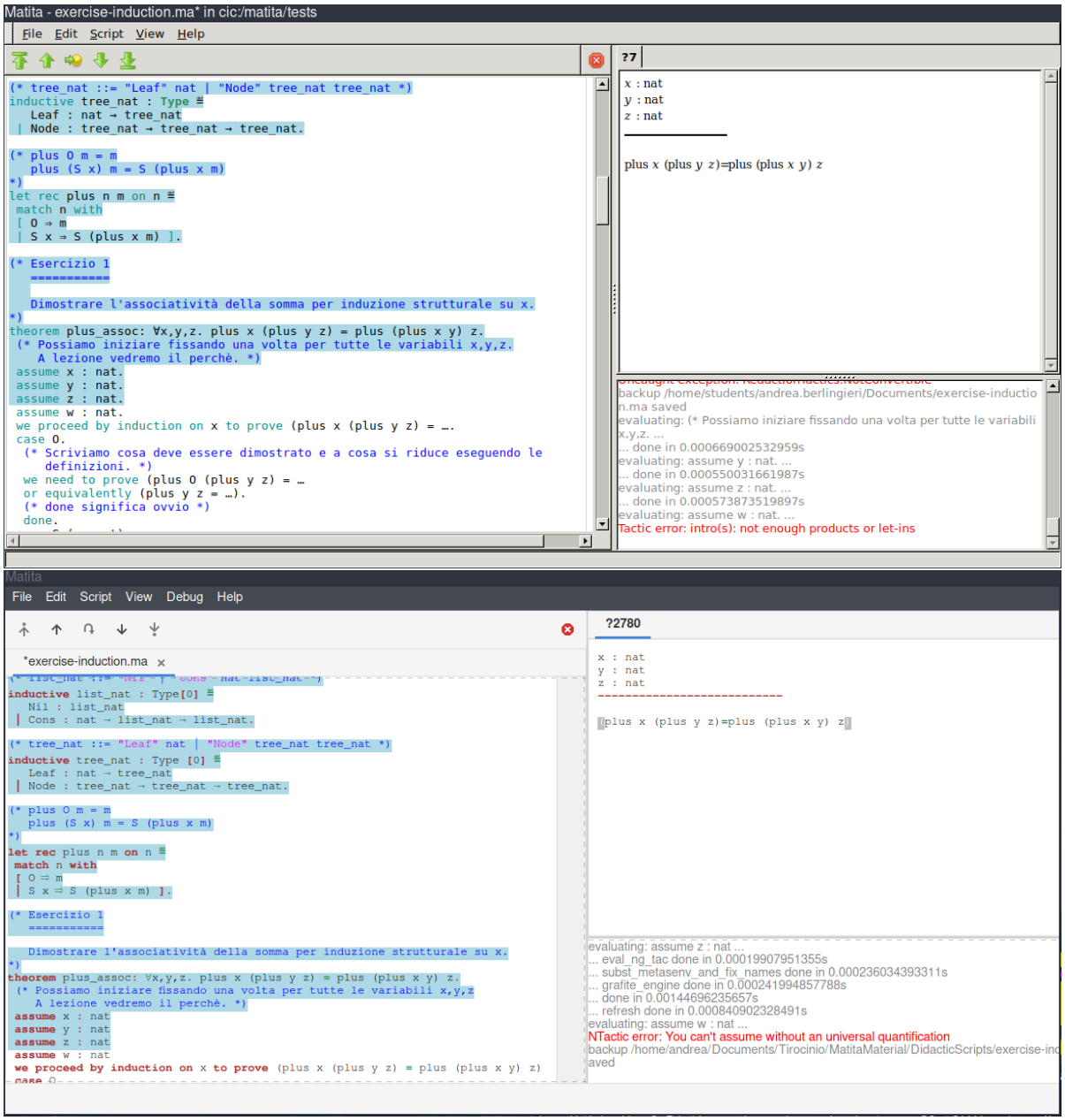

Figura 3.1: Confronto tra gli errori mostrati all'utente in caso di applicazione di **assume** senza quantificazione universale.

In entrambi i casi viene fatto un controllo sul tipo della conclusione del goal corrente, verificando che si tratti effettivamente di un prodotto. Successivamente viene controllato che il tipo fornito dall'utente sia alfa-equivalente al tipo del termine t legato dal prodotto.

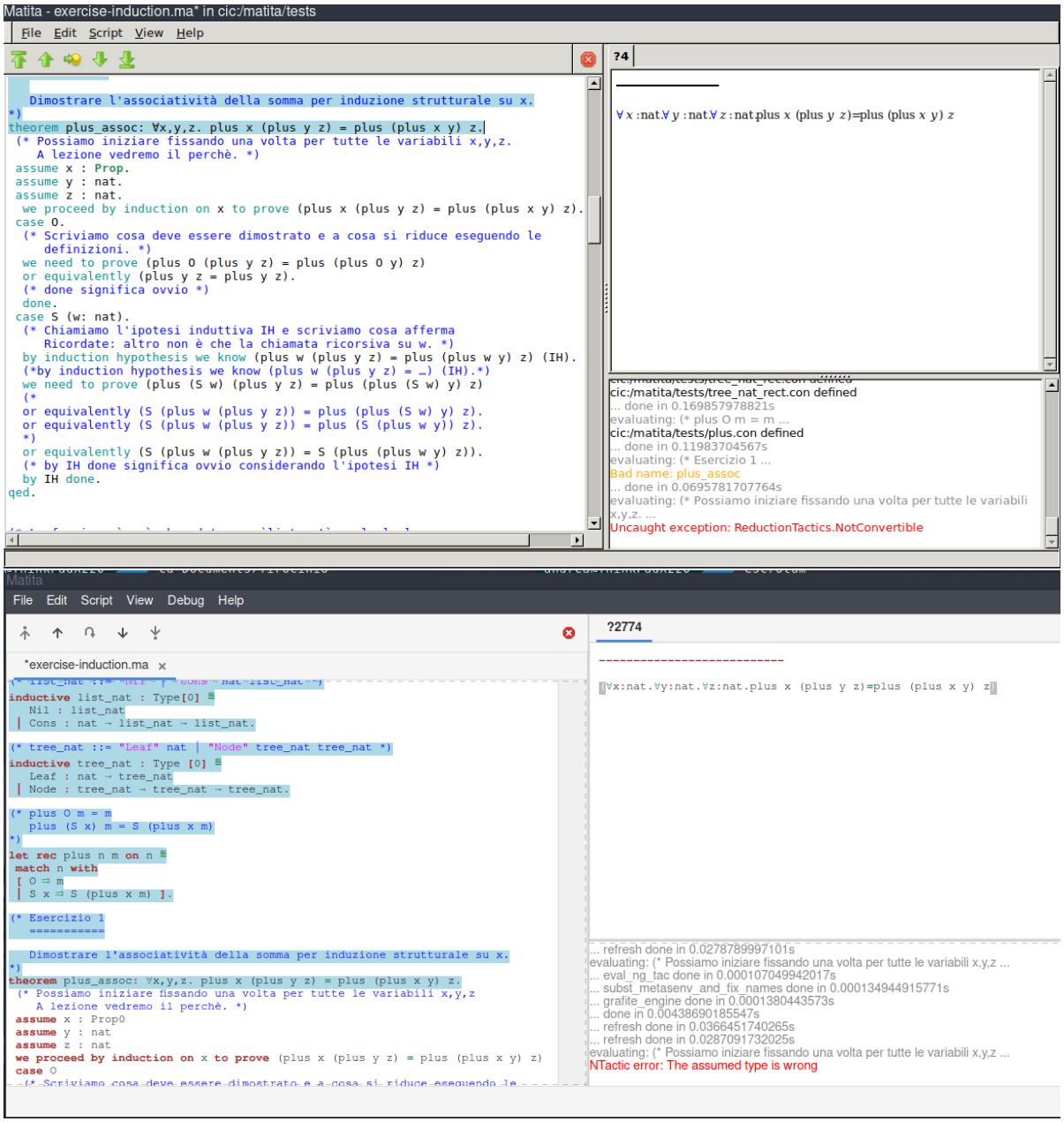

Figura 3.2: Confronto tra gli errori mostrati all'utente in caso di applicazione di **assume** con tipo errato.

Se questi controlli sono superati con successo alla metavariabile ?1 che rappresenta la conclusione corrente viene assegnato il termine $\lambda t : T.?2$, e la nuova conclusione da dimostrare diventa ?2, con la nuova ipotesi $t : T$ aggiunta al contesto della prova.

In Matita 0.5.x questi controlli erano fatti implicitamente dalle tattiche procedurali utilizzate per implementare assume e suppose. Di conseguenza, in caso di un errore, dovuto al fatto che la conclusione non fosse un prodotto o che il tipo fornito fosse sbagliato, l'errore mostrato all'utente era un errore di difficile comprensione, in quanto molto specifico nel contesto dell'implicazione delle tattiche procedurali. Nella nuova implementazione, poiché questi controlli vengono fatti direttamente, è possibile intercettare queste situazioni di errore e fallire mostrando all'utente un errore più "human friendly". Ad esempio, se un utente prova ad usare la tattica suppose su una conclusion che non è un'implicazione logica, il sistema risponderà dicendo che è necessario che la conclusione sia un'implicazione logica per applicare la tattica. Si può osservare un esempio dei nuovi errori nelle Figure 3.1 e 3.2.

Un'altra differenza con la vecchia implementazione e con [11] è dovuta al fatto che le eventuali conversioni di tipo, con le versioni estese con **that is equivalent to**, sono gestite in maniera diversa. In particolare, **that is equivalent to**, nella nuova implementazione, è diventata una tattica standalone, parsata indipendentemente, e basata sui contesti volatili (vedi 3.4.3). Dopo una assume o una suppose, viene aggiunto un parametro **volatile_newypo** alla parameter list, per gestire eventuali conversioni di tipo che seguono la tattica applicata. Dopo una assume o una suppose, è possibile cambiare il tipo dell'ipotesi con una catena di **that is equivalent to** subito successivi all'applicazione della tattica.

3.6 Let $x := t$

La tattica Let permette di introdurre una definizione locale all'interno della prova. La sintassi è la seguente:

let $x := t$

Si tratta di un caso particolare della versione procedurale della stessa tattica, in cui la definizione locale viene fatta nello scope della conclusione. La versione procedurale permette di specificare un pattern per un termine diverso dalla conclusione in cui introdurre la definizione locale di x. Di fatto, l'implementazione di questa tattica è stata

fatta nel Grafite Parser, dove, quando viene parsata la letin dichiarativa, viene creato, nella rappresentazione interna di Matita, l'AST di una letin procedurale, il cui pattern matcha l'intera conclusione.

3.7 Tattiche con giustificazioni e automazione

Nel linguaggio dichiarativo abbiamo delle tattiche che, per essere applicate, richiedono di essere "giustificate", fornendo una o più ipotesi che permettono di svolgere il passo di inferenza rappresentato dalla tattica. Tutte queste tattiche si basano sull'automazione di Matita, descritta in [8] e che ha due compiti: completare le prove dal punto vista formale, andando a colmare i dettagli che tipicamente vengono lasciati indietro durante le dimostrazioni informali (e.g. passi di catene di uguaglianza che richiedono l'applicazione di principi quali la simmetria dell'uguaglianza), e cercare prove in maniera automatica. Per gli scopi del linguaggio dichiarativo, la prima di queste funzioni è quella che viene sfruttata.

Una giustificazione può essere specificata con la clausola **by** o con la clausola **using**. Mentre by permette di specificare uno o più termini da combinare nella costruzione di una prova da parte dell'automazione, la clausola using permette di applicare alla conclusione un termine di prova fornito dall'utente. In un certo senso, la **using** lavora ad un livello più basso rispetto alla **by**.

3.7.1 By just we proved

La tattica **by just we proved** permette di cominciare una catena di statement **that is equivalent to**, per cambiare la conclusione in più passaggi, oppure permette di aggiungere una nuova ipotesi al contesto. La sintassi è la seguente:

$$\textit{just}\ \textbf{we proved}\ t\ [(\textit{id})]$$

id è il nome della nuova ipotesi t da aggiungere al contesto della prova, ed è opzionale.

Nel caso *id* venga tralasciato, l'implementazione della tattica consiste nel verificare che il termine t sia alfa-equivalente alla conclusione corrente, e, in tal caso, aggiungere

alla parameter list il contesto **beta_rewrite**, in modo che l'utente possa utilizzare lo statement **that is equivalent to** sulla conclusione.

Nel caso invece *id* sia presente, viene effettuato un taglio logico con il termine t. Sul nuovo goal "di taglio" viene utilizzata la giustificazione fornita dall'utente per chiuderlo; dopodiché, il sistema passa al goal che corrisponde alla conclusione prima dell'applicazione della tattica, e introduce la nuova ipotesi *id* di tipo t nel contesto della prova.

3.7.2 By just done

La tattica **done** permette di chiudere un goal, utilizzando la giustificazione fornita dall'utente. La sintassi è la seguente:

just **done**

Tipicamente questa tattica viene utilizzata come passaggio finale della dimostrazione di un'ipotesi di taglio, di un caso di una dimostrazione per induzione o per casi, o come passo finale della prova.

Come tutte le altre tattiche basate su giustificazione, l'implementazione è basata sull'automazione.

Questa tattica svolge un ruolo fondamentale nella nuova gestione dello stack dei tinycals in presenza di goal multipli.

3.7.3 By just let such that

La tattica **By just let such that** permette di effettuare l'eliminazione di un'ipotesi esistenziale: data un'ipotesi del tipo $\exists x : T.P(x)$, aggiunge al contesto un termine per cui vale l'ipotesi esistenziale eliminata. La sintassi è la seguente:

just **let** *id* : *type* **such that** *prop* (*H*)

L'implementazione della tattica consiste nell'applicare un taglio logico, con $\exists id : type.\ prop$ come ipotesi di taglio, giustificare l'ipotesi di taglio con *just*, passare alla dimostrazione della conclusione corrente, e introdurre un'ipotesi *id : type* e un'ipotesi H : *prop*.

Questa è un'altra tattica in cui è stato sfruttato il nuovo approccio di Matita 0.99.x per quanto riguarda l'istanziazione delle metavariabili mediante termini notazionali, ottenendo un codice più semplice e pulito per la stessa semantica implementata in Matita 0.5.x.

3.7.4 By just we have

La tattica **By just we have** permette di effettuare l'eliminazione di una congiunzione: data un'ipotesi del tipo $A \wedge B$, aggiunge al contesto della prova corrente un'ipotesi con tipo A ed un'ipotesi con tipo B. La sintassi è la seguente:

just **we have** A (*id1*) **and** B (*id2*)

L'implementazione della tattica consiste nell'applicare un taglio logico, con $A \wedge B$ come ipotesi di taglio, giustificare l'ipotesi di taglio con *just*, passare alla dimostrazione della conclusione corrente, e introdurre un'ipotesi $id1 : A$ e un'ipotesi $id2 : B$.

Anche per questa tattica è stato sfruttato il nuovo approccio di Matita 0.99.x per l'istanziazione delle metavariabili mediante termini notazionali.

3.8 We need to prove

Questa tattica permette di andare a cambiare la conclusione corrente, mediante beta-riduzione, o di effettuare un taglio logico. Si tratta di una tattica molto utilizzata negli script dichiarativi. La sintassi è la seguente:

we need to prove *prop* [(*id*)]

Il parametro *id* è opzionale, e nel caso non venga fornito questa tattica ha un comportamento analogo a quello della tattica **by just we proved** senza *id*. In caso *id* venga fornito, viene aperto un nuovo goal per l'ipotesi di taglio, ed il goal corrente viene modificato aggiungendo l'ipotesi *id : prop* al contesto. È possibile, dopo questa tattica, cominciare una serie di equivalenze **that is equivalent to**, che modificano la conclusione del goal corrente. Nel caso venga effettuato un taglio logico, andranno dimostrati due

Figura 3.3: Confronto tra catene di semplificazione in Matita 0.99.x e 0.5.x.

goal: uno che corrisponde all'ipotesi di taglio, e l'altro che corrisponde al goal su cui è stata applicata la tattica.

Per questa tattica svolge un ruolo fondamentale la nuova gestione dello stack dei tinycals. Nel caso di un taglio logico quando viene chiuso il goal dell'ipotesi di taglio il sistema passa alla dimostrazione del goal su cui è stata applicata la tattica.

Questa nuova implementazione ha un vantaggio interessante rispetto alla vecchia: nella vecchia implementazione, infatti, la tattica poteva essere estesa con uno statement **that is equivalent to** per fare un passo di beta riduzione sulla conclusione, andando a modificarla. Tuttavia, era possibile fare un solo passo di beta riduzione. Nel caso l'utente avesse voluto semplificare la conclusione in più passaggi sarebbe stato necessario creare uno statement **that is equivalent to** con il primo passaggio di semplificazione, eseguirlo, dopodiché tornare indietro di uno statement, commentare il primo passaggio di semplificazione e scrivere il secondo, eseguire lo statement, e così via, fino a giungere alla forma normale (o ad una forma soddisfacente per l'utente). Mediante la nuova gestione dello statement **that is equivalent to** come tattica standalone, è possibile scrivere una catena di riduzioni ed eseguirla un passo alla volta. È possibile osservare questa differenza in uno script in Figura 3.3.

Nella nuova implementazione è stato inoltre aggiunto un controllo di tipo su *prop*, nel caso senza *id*, rispetto alla conclusione del goal corrente.

3.9 Induzione e analisi per casi

Le dimostrazioni per induzione o per casi sono molto frequenti negli script dichiarativi. In generale una dimostrazione di questi due tipi viene gestita in maniera analoga: viene applicato il principio di eliminazione del tipo induttivo del termine su cui viene applicata l'induzione o l'analisi per casi, e vengono aperti un numero di goal pari al numero di costruttori del tipo induttivo. Matita genera, per ogni tipo induttivo definito, un lambda termine che corrisponde al principio di induzione (o eliminazione) per quel tipo, che può essere utilizzato in una prova mediante la tattica procedurale **elim**.

Dal punto di vista della semantica della tattica, si vuole implementare il seguente comportamento:

1. l'utente deve dimostrare tutti i casi aperti;

2. l'utente deve dichiarare quale caso sta per dimostrare, prima di passare a dimostrarlo;

3. l'utente può dimostrare i casi nell'ordine che ritiene opportuno;

4. l'utente non può uscire dall'induzione corrente prima di averla chiusa. Ad esempio, se l'utente va per casi su un termine, con due casi A e B, e nel caso A va di nuovo per casi su un nuovo termine, con due casi A' e B', l'utente non può passare al caso B prima di aver completato A', B', e A;

5. l'utente non può dichiarare un caso al di fuori di un'induzione o un'analisi per casi;

6. quando l'utente inizia un caso, è necessario che lo termini prima di passare al prossimo.

Nel vecchio Matita i punti 3, 5 e 6 di questa semantica non erano implementati; mancavano inoltre opportuni controlli di tipo per alcune tattiche. Sfruttando le nuove caratteristiche introdotte in questa implementazione del linguaggio dichiarativo, e in particolare la nuova gestione dello stack dei tinycals e la parameter list, è stata implementata questa semantica nella sua interezza.

3.9.1 We proceed by induction/cases

Queste due tattiche aprono una dimostrazione che procede per induzione o per analisi per casi su un termine t. La sintassi è la seguente:

we proceed by {**induction** | **cases**} **on** t **to prove** *prop*

La differenza tra una dimostrazione per casi ed una per induzione su un termine $t : T$ di una proposizione $P(t)$, sta nel fatto che nell'analisi per casi è richiesta una dimostrazione di $P(c)$ per ogni costruttore c del tipo T, mentre nella dimostrazione per induzione è richiesta una dimostrazione di $P(c)$ per ogni costruttore senza argomenti di tipo T (casi base), ed una dimostrazione per i costruttori con argomenti di tipo T (casi induttivi) con, a disposizione dell'utente, le ipotesi induttive. Ad esempio, se abbiamo un tipo induttivo per le proposizioni della logica proposizionale e procediamo per induzione

su un termine F, nel caso $F1 \wedge F2$ avremo a disposizione le ipotesi $P(F1)$ e $P(F2)$ per dimostrare $P(F1 \wedge F2)$.

Nell'implementazione della tattica nel caso dell'induzione, viene controllato che *prop* sia alfa-equivalente alla conclusione corrente. Dopodiché viene applicato il principio di eliminazione del tipo del termine t ad un numero adeguato di ? ed al termine t. Nel caso di questa implementazione il termine viene costruito nel codice, ma, sfruttando l'"intelligenza" di Matita, è possibile tralasciare alcuni dettagli del termine che verranno poi riempiti in maniera adeguata. Questi termini non specificati sono quelli contrassegnati da ?.

Il principio di induzione per un termine richiede un termine per ognuno degli argomenti del tipo (se il tipo è dipendente da termini), un termine per ognuno dei costruttori del tipo induttivo, ed il termine su cui il principio viene utilizzato. Il numero adeguato di argomenti impliciti da passare al principio di induzione del tipo è quindi dato dalla somma tra il numero di argomenti del tipo induttivo ed il numero di costruttori, più un ulteriore argomento implicito per la proposizione a cui viene applicato il principio di induzione. Queste informazioni sul tipo del termine vengono recuperate a livello di codice mediante interazione con il kernel di Matita.

Nella vecchia implementazione la struttura dati che contiene informazioni per un tipo induttivo, denominata *indtyinfo*, non conteneva la lista dei costruttori del tipo induttivo, ma solo il loro numero. Per implementare una corretta gestione dei casi è necessaria la lista dei costruttori, in quanto questa contiene i nomi dei costruttori, che vengono assegnati ai nuovi goal aperti dalla tattica. La struttura dati che contiene le informazioni sui tipi induttivi e la funzione che recupera queste informazioni sono state quindi modificate di conseguenza.

Come ultimo passo dell'implementazione, viene aggiunto un contesto adeguato alla lista dei parametri (*cases* in caso di analisi per casi, *induction* in caso di dimostrazione per induzione) e viene fatto un push di tutti i goal della lista Γ del livello appena precedente nello stack dei tinycals all'ultimo livello, nella lista τ. Poiché tutti i nuovi goal aperti si trovano in τ, e Γ risulta vuota nell'ultimo livello, Matita non pone il focus su alcun goal. In questo modo l'utente potrà, e dovrà, specificare un caso per proseguire nella dimostrazione.

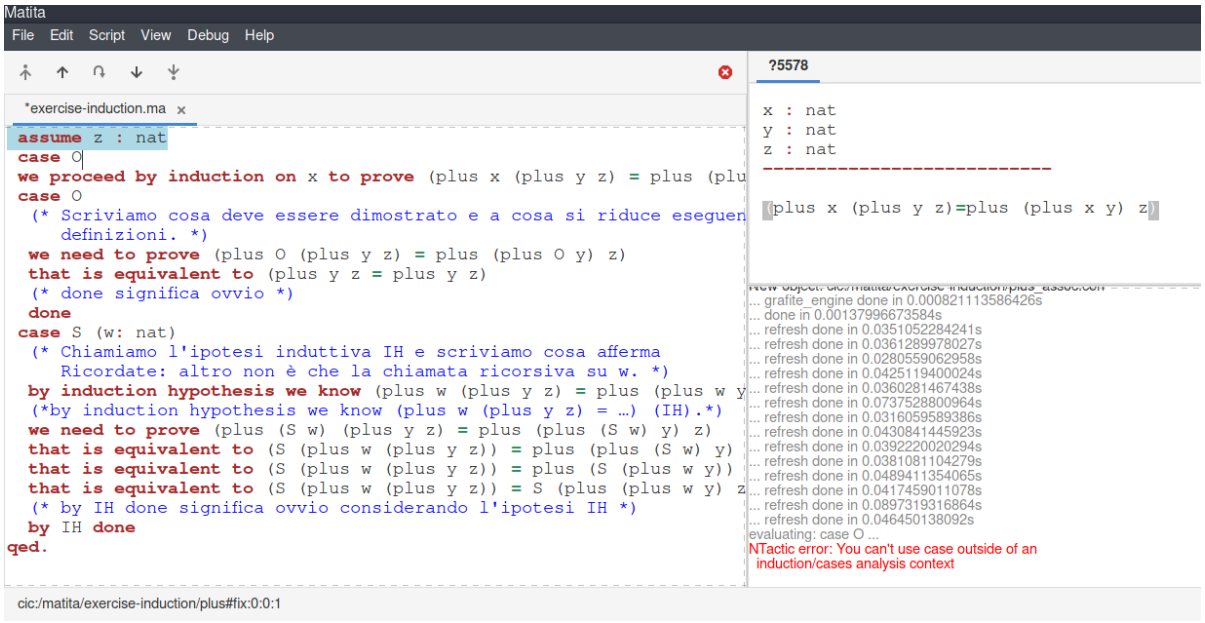

Figura 3.4: Errore mostrato in caso di applicazione di **case** fuori da un contesto di induzione/analisi per casi.

L'implementazione della **we proceed by cases** differisce dalla **we proceed by induction** in quanto la prima si basa sulla tattica procedurale corrispondente, la cases. Il motivo di questa scelta è dovuto al fatto che nel caso della cases non è stato necessario modificare il comportamento della tattica procedurale corrispondente, ed è stato quindi scelto di riutilizzarla. La **we proceed by induction** dichiarativa, invece, che corrisponde alla tattica di eliminazione procedurale, differisce da quest'ultima in quanto il principio di eliminazione viene applicato ad un numero preciso di argomenti, mentre nell'eliminazione procedurale il principio viene applicato a …, che sta ad indicare 0 o più argomenti impliciti. In questo modo, la we proceed by induction dichiarativa è meno soggetta ad errori dovuti al raffinamento di Matita dei termini.

Come ulteriore cambiamento nella nuova implementazione, viene mostrato un errore più human friendly nel caso il termine t sia diverso dalla conclusione corrente in cui viene cominciata la dimostrazione per induzione/casi.

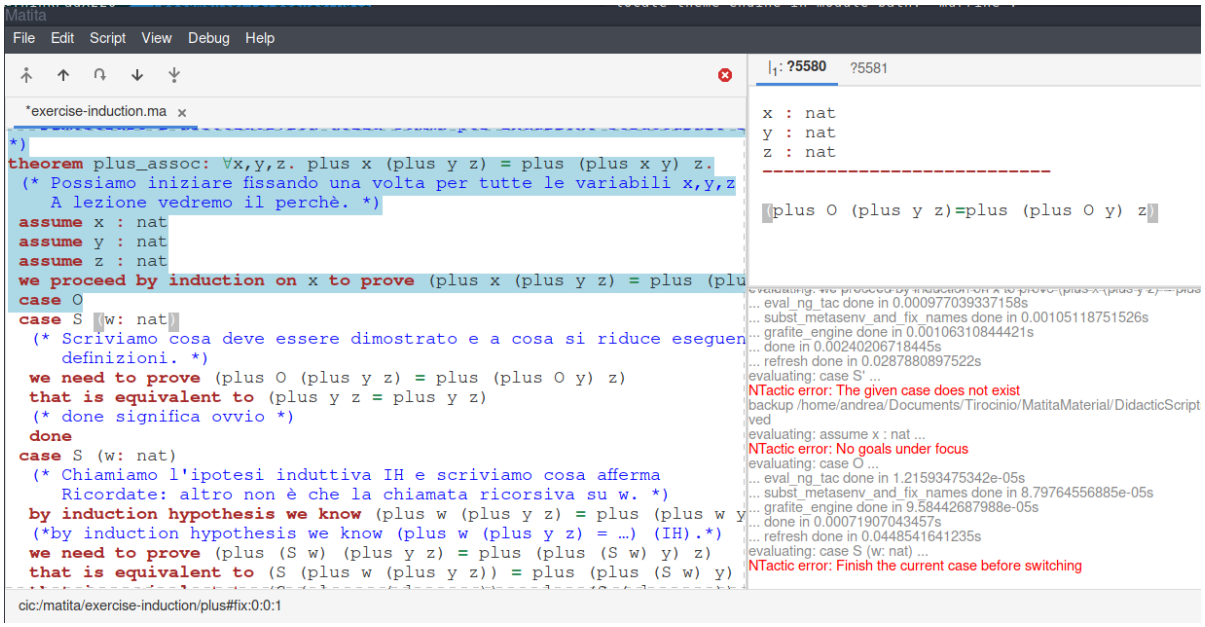

Figura 3.5: Errore mostrato in caso di applicazione di **case** senza aver completato il caso corrente.

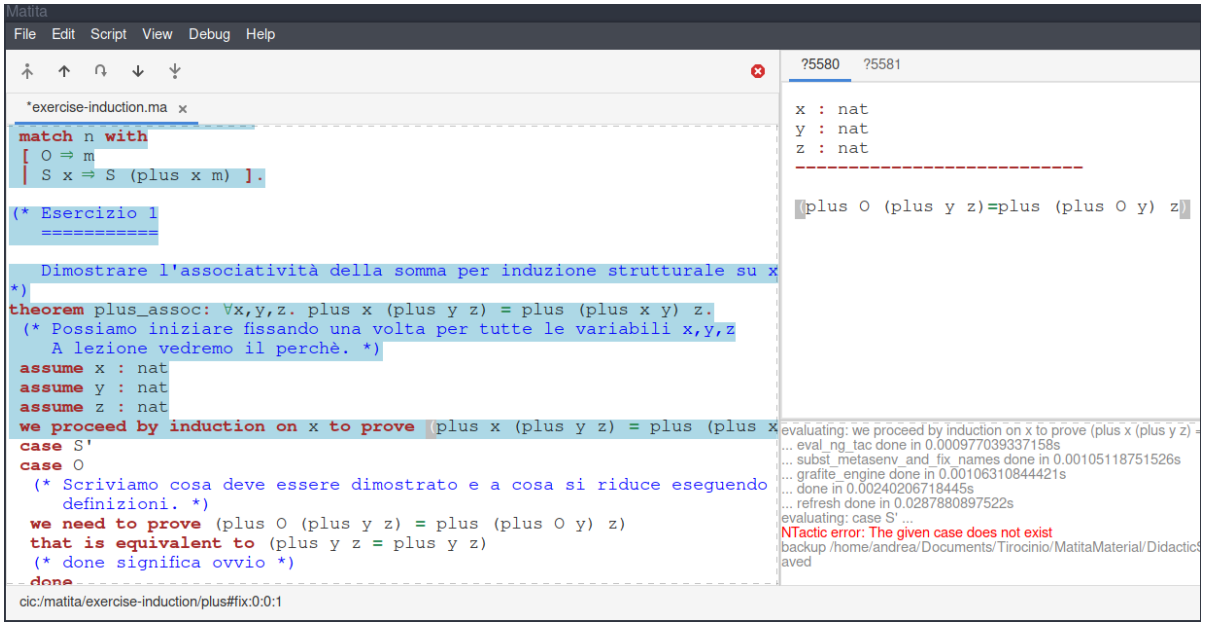

Figura 3.6: Errore mostrato in caso di applicazione di **case** con un costruttore inesistente per il tipo eliminato.

3.9.2 Case

Questa tattica permette di cominciare la dimostrazione di un caso di una dimostrazione per induzione o per casi. La sintassi è la seguente:

$$\textbf{case}\ id\ [arg_1 : T_1]\ \ldots\ [arg_n : T_n]$$

id è il nome di un costruttore del tipo induttivo del termine su cui è condotta la dimostrazione per induzione/casi, e $arg_1, \ldots, arg_n$ rappresentano gli argomenti del costruttore, accompagnati dal loro tipo. Per ognuno degli argomenti forniti, viene aggiunta un'ipotesi $arg_i : T_i$ al contesto corrente.

Nella nuova implementazione, la case sfrutta la parameter list del livello corrente dello stack dei tinycals per verificare che l'utente si trovi nel contesto di una dimostrazione per casi o induzione. Se questo non è il caso, un errore appropriato viene mostrato all'utente, come si può osservare in Figura 3.4. Inoltre, se il contesto è quello di una dimostrazione per casi o induzione e l'utente è nel mezzo della dimostrazione di un caso, la case fallisce con un errore che indica all'utente che deve finire la dimostrazione del caso corrente, come mostrato in Figura 3.5.

In assenza di condizioni di errore, la case pone il focus sul goal specificato dall'utente e introduce le ipotesi sugli argomenti del costruttore. Per sapere quale goal contiene la dimostrazione del caso richiesto viene sfruttato una struttura dati detta *metaattrs*, che permette di associare dell'informazione ai goal aperti. In particolare, durante l'apertura di una dimostrazione per casi o per induzione, ai goal aperti per ogni caso viene associato il nome del costruttore, che viene utilizzato poi per porre l'attenzione su un goal in particolare. La case cerca, tra i goal su cui porre il focus, solo nella lista τ dell'ultimo livello dello stack dei tinycals con la nuova gestione. Questo implica che non è possibile "cambiare induzione", da una più interna ad una più esterna, mediante la case, anche se i due tipi eliminati fossero gli stessi.

3.9.3 By induction hypothesis we know

Questa tattica permette di introdurre un'ipotesi di induzione in una dimostrazione per induzione. La sintassi è la seguente:

by induction hypothesis we know *prop* (H)

L'effetto è quello di introdurre un'ipotesi H : *prop* nel contesto della prova corrente. Di fatto, questa tattica è equivalente ad una **suppose** *prop* (H), ed infatti nell'implementazione vi è una fattorizzazione del codice in questo senso. Le ipotesi induttive sono presenti solo nei casi induttivi di una dimostrazione per induzione.

3.10 Catene di uguaglianze

Matita permette di costruire catene di uguaglianze, basate sulla transitività del predicato di uguaglianza, mediante una serie di tattiche che possono portare a dimostrare una conclusione del tipo $t_1 = t_k$ in maniera banale (e.g. $t = t$), o permettono di aggiungere un'ipotesi equazionale al contesto.

Nella vecchia implementazione la parte di codice che implementava questa parte è quella che è stata maggiormente modificata nel porting alla nuova versione di Matita. In particolare, sia il parsing che l'implementazione delle tattiche corrispondevano ad un solo blocco di codice, poco coeso e con varie parti molto simili. Nella nuova implementazione il parsing è stato separato per le tre tattiche, l'implementazione è stata fattorizzata e separata, e la semantica delle tattiche che cominciano una catena di uguaglianze è cambiata, in modo che queste tattiche comincino una catena di uguaglianze, e i successivi passi siano portati avanti dalla tattica di uguaglianza. Nell'implementazione vecchia, invece, per cominciare una catena di uguaglianze era necessario fornire anche il primo passo dell'uguaglianza, che veniva parsato ed eseguito assieme alla tattica che cominciava la catena.

Un'ulteriore novità di questa implementazione è legata al fatto che la tattica di uguaglianza non può essere utilizzata al di fuori di una catena di uguaglianze. Questa policy è stata implementata sfruttando la parameter list.

3.10.1 Conclude

La tattica **conclude** permette di cominciare una catena di uguaglianze sulla conclusione del goal correntemente aperto. La sintassi è la seguente:

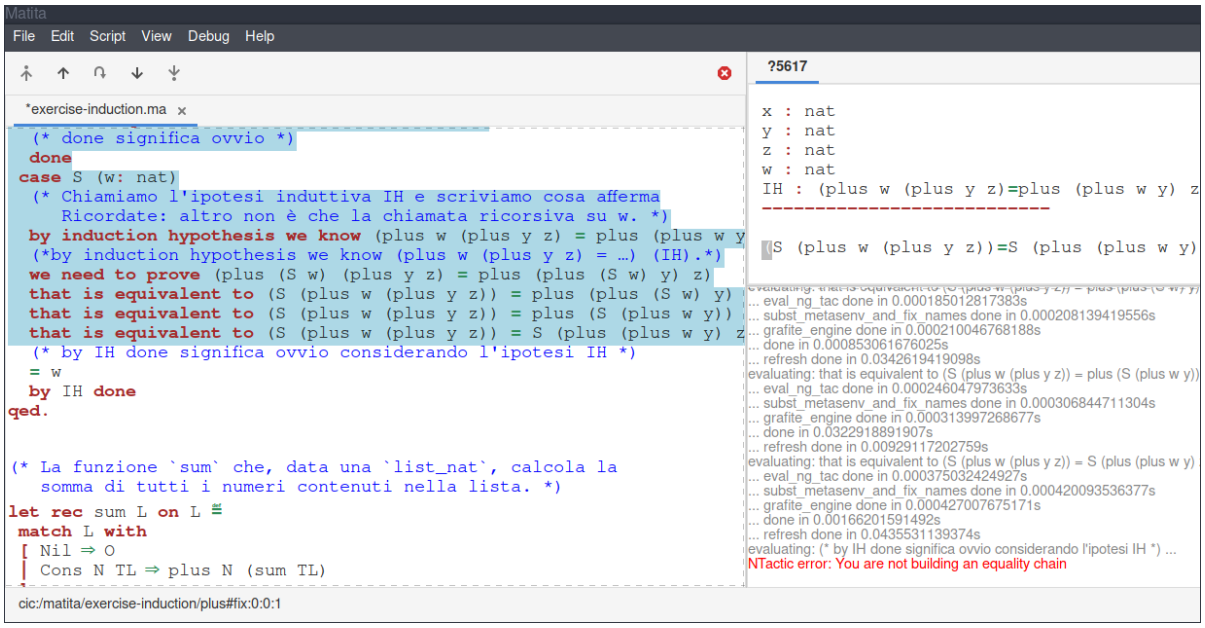

Figura 3.7: Errore mostrato in caso di applicazione di = al di fuori del contesto di una catena di uguaglianze.

conclude t_1

Se la conclusione corrente consiste di un'uguaglianza $t_1 = t_k$, questa tattica comincia una catena di uguaglianze il cui primo termine è t_1, ed il cui ultimo termine è t_k. Finita la catena, il goal corrente viene chiuso.

Nell'implementazione della tattica si controlla che la conclusione sia un'uguaglianza, e che il t_1 fornito dall'utente sia alfa-equivalente al t_1 che rappresenta il *left-hand-side* della conclusione corrente. In tal caso viene aggiunto un contesto volatile di *rewrite* alla parameter list del livello corrente dello stack dei tinycals, e risulta possibile proseguire la catena di uguaglianze. In caso la conclusione non sia un'uguaglianza, o il termine fornito non sia equivalente al *left-hand-side* della conclusione corrente, un errore appropriato viene mostrato all'utente.

3.10.2 RewritingStep

Questa tattica permette di proseguire una catena di uguaglianze, cominciata con la tattica conclude o con la tattica obtain. La sintassi è la seguente:

$$= t_i \ \textit{just} \ [\textbf{done}]$$

L'effetto di questa tattica è quello di applicare il principio di transitività dell'uguaglianza al termine fornito e al *left-hand-side* dell'uguaglianza corrente. Questo passo di uguaglianza deve essere giustificato mediante la *just* fornita dall'utente. Nel caso venga specificata la keyword **done**, questo passo di uguaglianza rappresenta l'ultimo e chiude il goal corrente.

Nell'implementazione della tattica si controlla che il contesto corrente sia di *rewrite*. In caso contrario, la tattica fallisce informando l'utente che non si trova nella dimostrazione di una catena di uguaglianze, come si può osservare in Figura 3.7. Dopodiché, vengono estratti dall'uguaglianza corrente i due lati dell'uguaglianza e viene applicato il principio di transitività dell'uguaglianza al tipo di t_i, al *left-hand-side* corrente (*plhs*), a t_i, ed al *right-hand-side* corrente (*prhs*).

Il goal che viene aperto per dimostrare che $plhs = t_i$ viene chiuso mediante la giustificazione fornita. Il goal corrente viene modificato in $t_i = prhs$, in accordo con il passo di uguaglianza appena eseguito.

3.10.3 Obtain

La obtain è analoga alla conclude, ma, invece di lavorare sulla conclusione, introduce una nuova ipotesi equazionale ad contesto, ed invece di condurre una catena di uguaglianze che porti da un t_1 di partenza al t_k che fa da *right-hand-side* dell'uguaglianza corrente, permette di dimostrare un'uguaglianza generica $t_1 =?$. La sintassi è la seguente:

$$\textbf{obtain} \ (\textit{id}) \ t_1$$

Nell'implementazione della tattica viene effettuato un taglio logico, con ipotesi di taglio $t_1 =?$. Verranno aperti due nuovi goal, uno per il nuovo termine ?, che è possibile ignorare, ed uno che contiene l'uguaglianza che si vuole aggiungere al contesto della prova.

Nel goal della prova viene introdotta un'ipotesi $t_1 =?$, dove ? verrà sostituito, una volta conclusa la catena di uguaglianze, con l'ultimo termine della catena. Il goal per ? verrà chiuso per side effect quando verrà conclusa la catena di uguaglianze. Viene aggiunto, alla parameter list, un nuovo contesto volatile di *rewrite*, e sarà possibile proseguire nella catena di uguaglianze mediante la tattica di uguaglianza. La catena di uguaglianza viene chiusa con uno statement **done**, o con un passo di uguaglianza terminante (i.e. che include un **done** finale).

Capitolo 4

Conclusioni e sviluppi futuri

4.1 Sviluppi futuri

Gli obiettivi all'inizio di questo lavoro erano due: il ripristino del linguaggio dichiarativo di Matita nella versione 0.99.x, ed il ripristino di una funzionalità che permetteva di ricostruire una prova dichiarativa a partire da un termine CIC che rappresenta una prova corretta. Per motivi di tempo, purtroppo, questa seconda parte del lavoro non è stata completata. Uno sviluppo futuro per questo progetto sarebbe il proseguimento del lavoro, sulla base della nuova implementazione del linguaggio dichiarativo e di Matita stesso. Infatti, la versione vecchia di Matita conteneva delle strutture dati e delle funzioni apposite per questa parte, che nella versione nuova non sono state riportate. Questi aspetti della vecchia implementazione andranno rielaborati per ripristinare nel nuovo Matita questa funzionalità.

4.1.1 Ricostruzione di una prova dichiarativa a partire da un termine CIC

L'idea è quella di ricostruire, a partire da un termine CIC e in base alla struttura del lambda termine, una sequenza di tattiche dichiarative applicabili, in modo da ottenere una versione dichiarativa di una prova già esistente. Un termine CIC di prova può provenire da varie fonti: ad esempio, può essere ottenuto da un theorem prover che utilizza lo

stesso calcolo logico di Matita per la rappresentazione dei termini di prova, come può essere Coq; oppure può venire da una libreria distribuita di teoremi formalizzati all'interno di Matita. In ogni caso, l'obiettivo è di ricostruire la struttura di una prova già esistente, migliorandone la leggibilità e manutenibilità mediante il linguaggio dichiarativo.

Questo processo di ricostruzione di una prova non è pensato per ricostruire identicamente una prova costruita mediante il linguaggio dichiarativo. Durante il processo dettagli pedanti della dimostrazione e alcuni passaggi possono essere soppressi nel risultato finale. Ciò che si vuole garantire è che questo processo abbia un risultato consistente, e raggiunga un punto fisso dopo un'iterazione: una prova ricostruita, ricompilata, e nuovamente ricostruita dovrebbe risultare identica.

4.1.1.1 Algoritmo di double type-inference

Una delle problematiche nella ricostruzione di una prova risiede nel riconoscere gli eventuali passi di conversione di tipo che sono avvenuti in una prova a partire dal lambda termine. Per ricostruire la prova si va a lavorare sui sottotermini del lambda termine di prova. Ad un sottotermine è possibile associare due tipi: uno è il tipo dovuto alla posizione che il sottotermine occupa nel termine; l'altro è il tipo legato alla struttura del sottotermine. In generale, questi due tipi corrispondono. Tuttavia, se questo non avviene, vi è stato un passo di conversione di tipo (e.g. **that is equivalent to**).

In Matita 0.5.x, oltre all'algoritmo di *type inference* del kernel, che dato un termine ne calcola il tipo, era presente un algoritmo di *double type inference*, ispirato da [12], che restituiva, per ogni termine, la sua sorta e due informazioni ulteriori: il *synthesized type*, e l'*expected type*. Il primo corrisponde al tipo calcolato dal tipico algoritmo di tipaggio del kernel, e deriva dall'utilizzo del sottotermine all'interno del termine di cui fa parte, mentre il secondo deriva dalla struttura del sottotermine. Queste informazioni venivano memorizzate all'interno di una HashTable, che metteva in relazione un termine con la sua sorta e i suoi due tipi.

Ora, associare un'informazione ad un termine non è banale, poiché uno stesso termine (e.g. x) può comparire più volte in contesti diversi e con significati diversi. Per associare un'informazione univoca ad un determinato sottotermine di un termine esistono varie soluzioni. Nella vecchia implementazione, le informazioni erano associate all'indirizzo

in memoria in cui era memorizzata la rappresentazione del termine in OCaml. Questo approccio era semplice, ma richiedeva una certa accortezza per garantire che uno stesso indirizzo non fosse usato per due termini diversi e non causare quindi dei mismatch. Un approccio più elegante, ma più costoso, è quello di associare le informazioni di tipaggio al path di un sottotermine all'interno di un termine. Questa strada sarebbe, preferibilmente, quella da intraprendere per il completamento del lavoro.

4.2 Automazione negli script dichiarativi

Varie tattiche dichiarative sono basate su giustificazioni, le quali, a loro volta, sono basate sul motore di automazione di Matita. Di particolare rilievo sono le giustificazioni fornite alla tattica **done** per chiudere un goal corrente.

Mentre, da un punto di vista procedurale, è desiderabile che l'automazione lavori al massimo delle sue capacità, nel caso dichiarativo questa "efficacia" dell'automazione può avere uno svantaggio notevole per le applicazioni del software nella didattica. Infatti, un motore di automazione troppo "intelligente" rischia di essere in grado di chiudere goal con giustificazioni minime, o a volte anche assenti. Quando ad uno studente viene sottoposto un esercizio di dimostrazione con Matita, ci si aspetta che lo completi capendo i passaggi della dimostrazione ed utilizzando le ipotesi ed i teoremi giusti al momento giusto. Se però lo studente scopre che la tattica **done** è in grado di risparmiargli del lavoro mentale nell'esercizio, c'è il rischio che gli esercizi vengano completati senza che vi sia un reale apprendimento dietro, bensì lo sfruttamento di un aspetto del sistema software.

Il motore di automazione di Matita non era stato progettato per essere "indebolito" per questo tipo di fini. Sebbene in questo lavoro di implementazione del linguaggio dichiarativo ci siano stati alcuni tentativi di rendere l'automazione meno potente per il linguaggio dichiarativo, questi indebolimenti hanno portato ad un'automazione incapace di svolgere passi di inferenza anche molto semplici. Per indebolire il motore di automazione si è tentato di impedire che questo calcolasse da solo i termini candidati per la conclusione della prova a partire dalle ipotesi locali e dai teoremi già dimostrati in precedenza. In questo modo il motore di automazione sarebbe stato costretto ad uti-

lizzare i soli candidati forniti dall'utente con il comando **by**, e per concludere la prova sarebbe stato necessario fornire i candidati corretti. Questi tentativi di indebolimento non hanno tuttavia prodotto i risultati attesi, e sono stati scartati nell'implementazione finale poiché avrebbero reso il frammento dichiarativo di Matita inutilizzabile per le dimostrazioni degli esercizi didattici.

Di conseguenza, uno sviluppo futuro che potrebbe partire da questo lavoro sarebbe la rivisita del motore di automazione, per includere la possibilità di "indebolirlo" per fini didattici, rendendolo sufficientemente intelligente da svolgere i necessari passi di inferenza date le ipotesi ed i teoremi giusti, ma non tanto intelligente da trovare la prova da solo. Questo lavoro potrebbe essere svolto nell'ambito dell'implementazione di alcune ottimizzazioni per l'implementazione del *given-clause algorithm* suggerite in [7].

4.3 Conclusioni

Il risultato finale di questo lavoro di tesi è stato un ripristino e miglioramento del linguaggio dichiarativo in Matita per la versione 0.99.x. Tra le ragioni che erano alla base di questo progetto vi erano la curiosità riguardo i principi alla base di un Interactive Theorem Prover, ed il desiderio personale di poter portare un contributo al progetto Matita per quanto riguarda le sua applicazioni didattiche. Personalmente trovo che i prossimi studenti dell'Università di Bologna potranno apprezzare le novità e le migliorie introdotte, e potranno esercitarsi e sperimentare con uno strumento software più avanzato e di più semplice utilizzo rispetto al passato.

Strumenti software utilizzati per la parte pratica della didattica, come può essere Matita o un compilatore per un linguaggio di programmazione, sono di fondamentale importanza per l'apprendimento, e trovo personalmente che il loro uso andrebbe esteso anche ad altre materie dell'Informatica. La possibilità di esercitarsi in autonomia e di avere un feedback istantaneo rispetto ai propri esperimenti e alla personale comprensione di un argomento rappresenta un'opportunità irrinunciabile per mettersi alla prova, trovare le lacune e le mancanze del proprio apprendimento, e spingersi oltre gli argomenti appresi. Molte materie di studio, e in particolare quelle materie con una forte componente teorica o con una componente pratica di difficile esercitazione, possono

essere affrontate con un'agevolazione maggiore attraverso il supporto di uno strumento software.

Un esempio interessante di materia il cui apprendimento gioverebbe di uno strumento software per il supporto dell'apprendimento potrebbe essere lo studio dei linguaggi di programmazione. All'università di Bologna questi argomenti sono affrontati in un corso denominato Linguaggi di Programmazione, che comprende una prima parte su Macchine Astratte, Compilatori, Interpreti, Analizzatori Lessicali e Semantici, ed una seconda parte sulle caratteristiche che ricorrono maggiormente nei linguaggi di programmazione odierni e passati, quali le regole di scope delle variabili, le tecniche di astrazione sui dati e sul flusso di esecuzione, le caratteristiche e l'implementazione di alcuni paradigmi di programmazione, quali quello ad oggetti, quello logico, e quello funzionale.

Facendo riferimento alla seconda parte, sarebbe interessante, da studente, avere uno strumento software che permetta di esercitarsi su un linguaggio di programmazione $\mathcal{L}$ la cui implementazione può essere configurata prima della compilazione del programma. In questo modo, lo studente potrebbe sperimentare gli effetti di una diversa regola di scope delle variabili, di un diverso tipo di passaggio dei parametri, di una diversa politica di binding delle funzioni di ordine superiore, di una diversa strategia di riscrittura nel paradigma funzionale, etc., senza dover cambiare linguaggio o ambiente di sviluppo.

Al momento, per apprezzare, ad esempio, le differenze tra uno scope statico ed uno scope dinamico, è necessario trovare un linguaggio con la prima caratteristica (e.g. C), una con la seconda (e.g. Perl), scrivere uno stesso programma nei due linguaggi, ed effettuare i test desiderati. Questo approccio richiede che lo studente sia in grado di trovare i linguaggi con le caratteristiche da lui richieste, che impari i linguaggi in questione e sia in grado di utilizzare i rispettivi ambienti di compilazione ed esecuzione. Per alcune caratteristiche dei linguaggi di programmazione il primo aspetto non è assolutamente banale, mentre il secondo e il terzo richiedono una certa quantità di tempo e lavoro. Uno strumento che risulti uniforme in tutti questi aspetti risulterebbe sicuramente più interessante a scopo didattico.

In conclusione, sebbene alcuni aspetti di questi lavoro siano rimasti incompiuti e siano stati lasciati come ulteriori sviluppi futuri, trovo che il risultato finale sia stato in ogni caso un ottimo progresso e spero risulti soddisfacente e utile ai futuri studenti

dell'Università di Bologna che affronteranno per la prima volta la Logica Matematica.

Appendices

Appendice A

Debugging

A.1 Gestione dei goal multipli dichiarativi con tattiche procedurali

La nuova gestione dei goal multipli negli script dichiarativi non è, in generale, compatibile con l'utilizzo dei tinycal **branch** e **shift** per la gestione dei goal sotto focus. Sebbene questo non abbia importanza quando si lavora con uno script puramente dichiarativo, può essere utile a volte, per motivi di debugging, andare ad agire sui termini di prova mediante le tattiche procedurali in uno script dichiarativo.

Per tornare ad uno stack gestito in maniera tradizionale e poter applicare tattiche procedurali è possibile utilizzare il tinycal **merge**. Infatti, merge rimuove un livello dallo stack, e aggiunge tutti i goal rimasti aperti nell'ultimo livello alla lista Γ del livello subito precedente. Per verificare lo stato dello stack in un qualsiasi momento, è possibile utilizzare il comando **print_stack**, che stampa su standard output il pretty printing dello stack. Quando tutti i goal di interesse sono presenti nella lista Γ dell'ultimo livello dello stack, si può procedere a gestire i goal in maniera tradizionale con i tinycal **branch** e **shift**.

In alternativa, è possibile utilizzare le tattiche procedurali, tenendo conto che la loro semantica dipende dallo stato corrente dello stack. In particolare, la **shift** non funziona se nel livello dello stack precedente a quello corrente non vi sono goal nella lista Γ, e,

se si vuole porre il focus su dei goal della lista τ del livello corrente, è necessario prima applicare il tinycal **merge**.

A.2 Pretty printing di AST

Parte del lavoro eseguito è stata la scrittura del codice per il pretty printing delle tattiche dichiarative. Per testare che un AST pretty printed fosse effettivamente riparsabile, è stato aggiunto del codice al Matita Engine per scrivere l'output del pretty printing di un teorema durante il parsing di uno script dichiarativo in un file con lo stesso nome dello script parsato, con l'aggiunta dell'estensione *.parsed.ma*. Non viene ristampato l'intero script poiché parte del codice di pretty printing è presente solo a scopo di testing, e non vi è garanzia che l'output ottenuto sia riparsabile per alcune parti dello script (e.g. le definizioni di funzioni ricorsive non hanno un pretty printing riparsabile al momento).

Il codice di pretty printing viene eseguito solo se uno script dichiarativo è "compilato" con *matitac* con l'argomento `-v`. Per testare che l'output del pretty printing sia effettivamente parsabile è possibile fare un copia-incolla in Matita interattivo, e verificare che lo script venga parsato senza problemi.

Bibliografia

[1] The coq proof assistant. `https://coq.inria.fr/`.

[2] Logic for computable functions. `https://howlingpixel.com/i-en/Logic_for_Computable_Functions`.

[3] The matita proof assistant. `https://matita.cs.unibo.it/`.

[4] The mizar proof assistant. `http://mizar.org/`.

[5] Proofs as programs. `http://nuprl.org/`.

[6] A.Asperti, W.Ricciotti, C.Sacerdoti Coen, and E.Tassi. A new type for tactics. In *In proceedings of ACM SIGSAM PLMMS 09, ISBN 978-1-60558-735-6*, pages 27–50, 2009.

[7] Andrea Asperti and Enrico Tassi. Superposition as a logical glue. In *Proceedings Types for Proofs and Programs, Revised Selected Papers, TYPES 2009, Aussois, France, 12-15th May 2009.*, pages 1–15, 2009.

[8] Andrea Asperti and Enrico Tassi. Smart matching. In *Intelligent Computer Mathematics, 10th International Conference, AISC 2010, 17th Symposium, Calculemus 2010, and 9th International Conference, MKM 2010, Paris, France, July 5-10, 2010. Proceedings*, pages 263–277, 2010.

[9] Hendrik Pieter Barendregt, Wil Dekkers, and Richard Statman. *Lambda Calculus with Types*. Perspectives in logic. Cambridge University Press, 2013.

[10] Claudio Sacerdoti Coen. Declarative representation of proof terms. *J. Autom. Reasoning*, 44(1-2):25–52, 2010.

[11] Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. Tinycals: Step by step tacticals. *Electr. Notes Theor. Comput. Sci.*, 174(2):125–142, 2007.

[12] Yann Coscoy. A natural language explanation for formal proofs. In *Logical Aspects of Computational Linguistics, First International Conference, LACL '96, Nancy, France, September 23-25, 1996, Selected Papers*, pages 149–167, 1996.

[13] Jean-Yves Girard. *Proofs and Types*. Cambridge University Press, 1989.

[14] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.