Università degli Studi di Bologna

Facoltà di Scienze Matematiche, Fisiche e Naturali

Corso di Laurea Specialistica in Informatica

Materia di Tesi: Informatica Teorica

Progettazione e realizzazione di una tattica di dimostrazione automatica basata su paramodulazione per il proof-assistant Matita

Tesi di Laurea di:
Alberto Griggio

Relatore:
Chiar.mo Prof. Andrea Asperti

II Sessione

Anno Accademico 2004-2005

Indice

Elenco delle figure

Capitolo 1

Introduzione

HELM[1] (Hypertextual Library of Mathematics) è un progetto a lungo termine, intrapreso dal prof. Andrea Asperti e il suo gruppo di ricerca, avente come obiettivo l'integrazione di diversi strumenti di supporto al ragionamento formale per creare, gestire e sfruttare un'ampia libreria ipertestuale di matematica formalizzata.

Uno dei componenti principali di HELM è il suo *proof-assistant*, Matita, mediante il quale è possibile estendere la libreria definendo nuovi teoremi e assiomi. Matita si basa su un framework logico di ordine superiore dotato di grande espressività, un'estensione del λ-calcolo tipato chiamata CIC[Sac04].

Per dimostrare un teorema (un *goal*), l'utente di Matita applica ad esso una sequenza di *tattiche*, che sostituiscono il goal iniziale con alcuni nuovi sottogoal, ai quali si applicano ancora delle tattiche, e così via finché non rimane nessun goal aperto. Il compito del proof-assistant è quindi quello di assicurare la correttezza dei vari passaggi, ma l'intera procedura di dimostrazione richiede l'intervento continuo dell'utente, che deve decidere ad ogni passo che tattica applicare.

Sembra pertanto chiaro che, da un lato, la necessità di seguire passo-passo il sistema anche per dimostrare sottogoal "banali" può facilmente diventare noioso e far perdere molto tempo, e che dall'altro lato ci possono invece essere dei teoremi la cui dimostrazione non è immediata, e che quindi richiedono diversi tentativi prima di essere dimostrati correttamente.

[1] `http://helm.cs.unibo.it`

Per questi motivi, Matita mette a disposizione una tattica `auto`[Sel04] che, dato un goal, tenta di trovarne una dimostrazione *automaticamente*, mediante l'applicazione dei teoremi e degli assiomi presenti nella libreria e delle ipotesi locali al goal stesso, senza richiedere l'intervento dell'utente.

Tuttavia, `auto` ha un problema: la presenza del predicato di uguaglianza la rende particolarmente inefficiente. Ciò è dovuto al modo in cui l'uguaglianza è definita all'interno della libreria di HELM, ovvero da una serie di assiomi di congruenza, la cui applicazione porta alla generazione di un numero eccessivamente alto di sottogoal, facendo così esplodere lo spazio di ricerca.

Lo stesso problema è stato affrontato con successo nell'ambito del theorem-proving automatico per la logica del primo ordine, [BG01], dove la soluzione adottata è stata quella di riservare un trattamento particolare al predicato di uguaglianza, e cioè di considerarlo come parte del linguaggio[2], con regole di inferenza dedicate, la più importante delle quali è chiamata *paramodulazione*.

L'obiettivo di questa tesi è stato quindi quello di investigare la possibilità di applicare la stessa tecnica nell'ambito di HELM e del suo calcolo di ordine superiore CIC, per implementare una nuova versione di `auto` che trattasse in maniera più efficiente l'uguaglianza. Il resto della tesi è dedicato ad illustrare in che modo ciò sia stato fatto.

1.1 Organizzazione della tesi

Nel Capitolo 2 vengono introdotti HELM, Matita e CIC, evidenziandone principalmente gli aspetti rilevanti ai fini di questa tesi.

Il Capitolo 3 presenta le basi teoriche della paramodulazione nell'ambito della logica del primo ordine con uguaglianza.

I successivi Capitoli 4 e 5 sono dedicati all'implementazione della tecnica di paramodulazione nell'ambito di HELM: in particolare, il Capitolo 4 è una descrizione ad alto livello dell'algoritmo principale, mentre il Capitolo 5 descrive gli aspetti critici per le prestazioni e le ottimizzazioni implementate.

Nel Capitolo 6 viene discussa la costruzione delle prove come termini ben

[2] si parla quindi, appunto, di logica del primo ordine con uguaglianza

tipati di CIC, fattore cruciale per l'integrazione con Matita, che è oggetto del successivo Capitolo 7.

Infine, il Capitolo 8 conclude presentando i risultati ottenuti e i possibili sviluppi futuri.

Capitolo 2

HELM, CIC e Matita

Questo capitolo presenta una visione del contesto in cui il lavoro di questa tesi si colloca. L'obiettivo qui non è di fornire una descrizione completa o tantomeno dettagliata del progetto HELM, ma piuttosto di mettere in rilievo gli aspetti che hanno influito più direttamente nella progettazione e nell'implementazione della tattica di paramodulazione. Per una trattazione approfondita, si veda [Sac04]. La prima parte è dedicata ad una panoramica generale sul progetto HELM e la sua architettura, mentre le rimanenti due introducono il sistema logico CIC e il proof-assistant Matita rispettivamente.

2.1 Una panoramica su HELM

Come già accennato nell'Introduzione, l'obiettivo principale del progetto HELM è la creazione di una serie di strumenti per lo sviluppo e lo sfruttamento di larghe librerie ipertestuali e distribuite di conoscenza matematica formalizzata, che comprendano tutto il materiale già codificato nei sistemi attuali.

L'architettura generale di HELM è riassunta in Figura 2.1. Vi si possono riconoscere quattro livelli ben distinti:

- Al centro troviamo la libreria di teoremi e assiomi, composta di documenti e metadati memorizzati in formato XML. La sua posizione centrale riflette bene l'approccio *document-centric* di HELM, in cui l'enfasi non è posta tanto sulle applicazioni per l'accesso ai dati, ma piuttosto sui dati stessi.

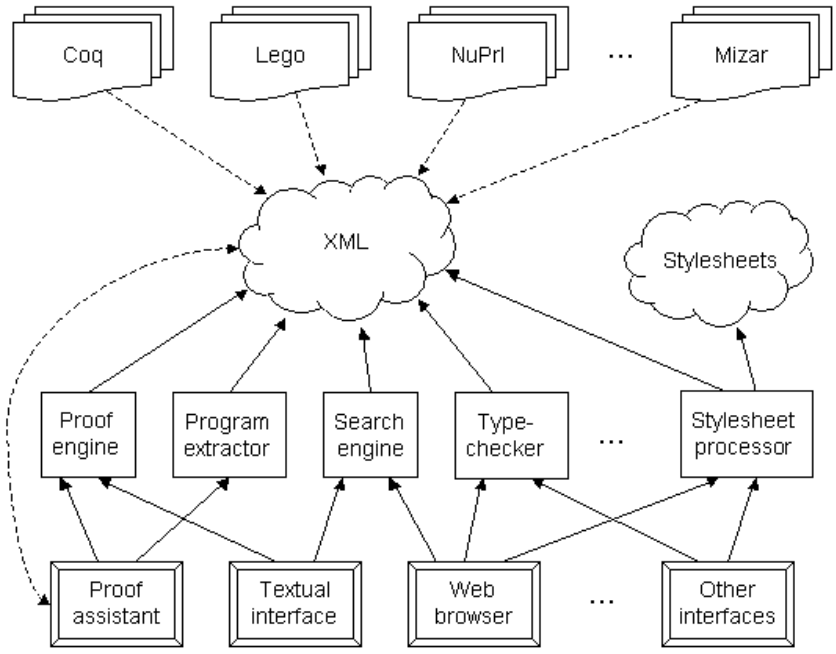

Figura 2.1: Architettura generale di HELM

- Immediatamente sopra nella figura sono elencati alcuni degli attuali proof-assistant: i collegamenti con la libreria indicano che quest'ultima contiene l'informazione proveniente dalle librerie di tali sistemi, opportunamente convertita al formato XML di HELM. Questo è uno degli aspetti caratterizzanti del progetto: l'integrazione e lo sfruttamento dell'informazione già codificata nei proof-assistant esistenti, per offrire agli utenti un database il più ricco possibile, anche a costo di una certa ridondanza.

- Al terzo livello si trovano i componenti "di basso livello" in grado di lavorare sul formato XML. Essi implementano in maniera modulare tutti gli eterogenei servizi messi a disposizione da HELM: in particolare, quelli fornite usualmente nei proof-assistant, come type-checking e type-inference, verranno analizzate più dettagliatamente nella prossima Sezione.

  Una classe importante di tool è quella responsabile della trasformazione dei documenti in un formato di resa appropriato: è durante questa trasformazione che deve avvenire il processo di rimatematizzazione del contenuto, basato sulla scelta, da parte dell'utente, di un'opportuna notazione.

- L'ultimo livello in basso è quello delle interfacce, che vengono costruite assemblando uno o più moduli del livello superiore, fornendone un accesso uniforme. Esempi di interfacce sono l'interfaccia web WHELP, che permette, attraverso un comune browser web, di navigare nella libreria, di effettuare ricerche, controllare termini e visualizzare assiomi, definizioni, teoremi e le loro relative dimostrazioni, ed il proof-assistant Matita, oggetto dell'ultima Sezione di questo Capitolo.

2.2 CIC - Calcolo delle Costruzioni (Co)Induttive

Questa Sezione tratta le caratteristiche fondamentali del framework logico utilizzato da HELM, cioè il *Calcolo delle Costruzioni (Co)Induttive* (CIC). Si tratta di un'estensione di ordine superiore del λ-calcolo tipato [Bar92], introdotto dal proof-assistant Coq [Coq04], che permette la definizione di tipi funzionali e (co)induttivi, nonché l'astrazione di termini su tipi, di tipi su tipi e di tipi su termini (tipi dipendenti), tutte caratteristiche che lo rendono estremamente espressivo: è possibile infatti (ed è stato effettivamente fatto nella libreria standard del proof-assistant Coq) codificare in esso teorie matematiche molto complesse, come ad esempio la Teoria degli Insiemi di Zermelo-Fraenkel.

Nel seguito, abbiamo seguito l'impostazione data da [Sac04] per la presentazione di CIC, operando tuttavia alcune semplificazioni ed omettendo molti dei dettagli, ed in qualche caso anche alcune parti del calcolo ritenute non necessarie nel contesto di questa tesi. In particolare, le porzioni di CIC riguardanti l'analisi per casi e la definizione di termini per (co)punto fisso sono solamente accennate nella descrizione della sintassi, e completamente omesse nella presentazione delle regole di convertibilità e di tipaggio. Si rimanda pertanto alla citata tesi di Sacerdoti Coen per un'esposizione completa.

Il resto della Sezione è diviso in due parti: nella prima vengono introdotte la sintassi e le regole di riduzione, di convertibilità e di tipaggio di CIC, mentre nella seconda sono discusse alcune estensioni al calcolo adottate nell'ambito di HELM.

t	$::=$	x	identificatori
	$\mid$	c	costanti
	$\mid$	i	tipi (co)induttivi
	$\mid$	k	costruttori di tipi (co)induttivi
	$\mid$	$\textsc{Set} \mid \textsc{Prop} \mid \textsc{Type}(j)$	sort
	$\mid$	$t\ t$	applicazione
	$\mid$	$\lambda x : t.t$	λ-astrazione
	$\mid$	$\Pi x : t.t$	prodotto dipendente
	$\mid$	$< t > CASES_i\ t\ OF$	analisi per casi
		$\quad k_1\ \overline{x} \Rightarrow t \mid \ \ldots\ \mid k_n\ \overline{x} \Rightarrow t$	
		END	
	$\mid$	$FIX_l\ \{x/n_1 : t := t\ ;\ \ldots\ ;\ x/n_m : t := t\}$	definizione per punto fisso
	$\mid$	$COFIX_l\ \{x : t := t\ ;\ \ldots\ ;\ x : t := t\}$	definizione per co-punto fisso

Tabella 2.1: Sintassi dei termini di CIC

2.2.1 La sintassi

Siano[1]

$\mathcal{V}$ una famiglia numerabile di nomi di variabili, $x, y, z, \ldots \in \mathcal{V}$.

$\mathcal{C}$ una famiglia numerabile di nomi di costanti, $c, c_1, \ldots, c_n \in \mathcal{C}$.

$\mathcal{I}$ una famiglia numerabile di nomi di tipi (co)induttivi, $i, i_1, \ldots, i_n \in \mathcal{I}$.

$\mathcal{K}$ una famiglia numerabile di nomi di costruttori di tipi (co)induttivi, $k, k_1, \ldots, k_n, k_1^1, \ldots, k_{n_1}^1, \ldots, k_1^m, \ldots, k_{n_m}^m \in \mathcal{K}$.

I termini ben formati, che verranno indicati con t, f, u, T, U, N, M, a seconda dell'uso che ne verrà fatto, ovvero dell'interpretazione che verrà data loro, sono definiti induttivamente nella tabella 2.2.1, dove $m, n_1, \ldots, n_m, j$ e l sono interi positivi, l è al più uguale al numero di funzioni definite nel proprio (CO)FIX e $\overline{x}$ è una sequenza, eventualmente vuota, di identificatori, la cui lunghezza

[1]In tutta la Sezione, con un piccolo abuso di notazione, le successioni di qualsiasi genere indiciate da 1 a n potranno essere vuote, salvo indicazioni contrarie.

verrà indicata con $|\overline{x}|$. Con $s, s_1, \ldots, s_n$ verranno indicate le sort. Si utilizzerà la notazione $T_1 \to T_2$ per $\Pi x : T_1.T_2$ quando x non compare libera in T_2. Il tipo induttivo i nel CASES è ridondante, in quanto tipi induttivi distinti hanno costruttori distinti, e viene indicato per maggiore chiarezza.

Si noti che, in CIC, non esistono distinzioni fra tipi e termini.

Le parentesi tonde verranno utilizzate per rendere concreta la sintassi astratta, con l'usuale convenzione per la quale l'applicazione è associativa a sinistra e la λ-astrazione a destra. Nella λ-astrazione e nei tipi dipendenti la x è legata nei corpi, ovvero dopo il punto. Analogamente, le x sono legate nelle definizioni per (co)punto fisso nei corpi delle funzioni del proprio blocco, ovvero dopo i simboli di assegnazione, e dopo le doppie frecce nella definizione per casi. Le definizioni di variabili libere, di α-conversione e di sostituzione sono quelle usuali.

2.2.2 Le riduzioni e la convertibilità

Prima di dare le regole di riduzione di CIC, è necessario definire i contesti e gli ambienti.

Contesti e ambienti

Un *contesto* Γ è una lista ordinata di dichiarazioni di variabili (e loro tipi) e prende la forma $[(x_1 : T_1), \ldots, (x_n : T_n)]$ dove le x_i sono tutte distinte. Per indicare che $(x : T)$ è un elemento di Γ scriveremo $(x : T) \in \Gamma$. $x \in \Gamma$ significa $\exists T.(x : T) \in \Gamma$. [] denota il contesto vuoto.

Un *ambiente* E è una lista ordinata di dichiarazioni di (tipi e corpi di) costanti e di tipi mutuamente (co)induttivi e prende la forma $[\omega_1, \ldots, \omega_n]$ dove ω_j ha la forma $c : T := t$ (dichiarazione di costante c) oppure la forma

$$\begin{array}{l}\Pi x_1 : U_1 \ldots \Pi x_h : U_h. \\ \quad \{i_1 : A_1 := \{k_1^1 : C_1^1 \ ; \ \ldots \ ; \ k_{n_1}^1 : C_{n_1}^1\} \ ; \\ \quad \ldots \ ; \\ \quad i_m : A_m := \{k_1^m : C_1^m \ ; \ \ldots \ ; \ k_{n_m}^m : C_{n_m}^m\} \\ \quad \}_{(CO)IND}\end{array}$$

(dichiarazione del blocco di tipi mutuamente (co)induttivi $i_1, \ldots, i_m$, dipendenti dai parametri $x_1, \ldots, x_h$, i cui costruttori sono $k_1^1, \ldots, k_{n_m}^m$) dove $x_1, \ldots, x_h$ sono

legati in $A_1, \ldots, A_m, C_1^1, \ldots, C_{n_m}^m$ e $i_1, \ldots, i_m$ possono apparire in $C_1^1, \ldots, C_{n_m}^m$ e $(CO)IND$ deve essere IND per i tipi induttivi e $COIND$ per quelli coinduttivi.

Tutti i nomi delle costanti definite in un ambiente devono essere distinti, così come quelli dei tipi mutuamente (co)induttivi e dei loro costruttori. Le notazioni per l'appartenenza e l'ambiente vuoto sono le stesse usate per i contesti. Infine, $i \in E$ e $k \in E$ sono definiti in maniera ovvia.

Le regole per l'introduzione di definizioni nell'ambiente e nei contesti verranno date nel paragrafo 2.2.3 essendo mutuamente definite con le regole di tipaggio di CIC.

β-riduzione, δ-riduzione, ι-riduzione e unfolding (o espansione)

In CIC è necessario definire quattro tipi di riduzioni, chiamate β-riduzione, ι-riduzione, δ-riduzione e unfolding (o espansione). Le prime due corrispondono all'eliminazione dei tagli per l'implicazione ed i tipi induttivi. La terza è standard nello studio della semantica dei linguaggi di programmazione. La quarta si incontra, in forme talvolta meno ristrette, nei linguaggi di programmazione lazy. Quest'ultima tuttavia, essendo ristretta ad i termini (co)induttivi, non verrà qui illustrata.

Definizione 2.1 (β-riduzione). È quella standard: $(\lambda x : T.M)N \rhd_\beta M\{N/x\}$ con la side condition, usualmente lasciata implicita, che le variabili libere di N non siano catturate in $M\{N/x\}$.

Definizione 2.2 (δ-riduzione). È quella standard: nell'ambiente E contenente la dichiarazione $c : T := t$ si ha $c \rhd_\delta t$.

Definizione 2.3 (ι-riduzione). Equivale alla β-riduzione per i tipi (co)induttivi. Nell'ambiente E contenente la dichiarazione

$$\begin{array}{l}\Pi x_1 : U_1 \ldots \Pi x_h : U_h. \\ \quad \{i_1 : A_1 := \{k_1^1 : C_1^1 \ ; \ \ldots \ ; \ k_{n_1}^1 : C_{n_1}^1\} \ ; \\ \quad \ldots \ ; \\ \quad i_m : A_m := \{k_1^m : C_1^m \ ; \ \ldots \ ; \ k_{n_m}^m : C_{n_m}^m\} \\ \quad \}_{(CO)IND}\end{array}$$

si ha

$$< T > CASES_{i_l}\ (k_j^l t_1 \dots t_h t'_1 \dots t'_{n_j})\ OF$$
$$\quad k_1^l\ \overline{x^1} \Rightarrow f_1 \mid \dots \mid k_n^l\ \overline{x^n} \Rightarrow f_n \qquad \rhd_\iota f_j\{t'_1/x_1\ ;\ \dots\ ;\ t'_{n_j}/x_{n_j}\}$$
$$END$$

dove $l \in \{1, \dots, m\}$, $\forall l \in \{1, \dots, n_j\}.|x_l| = n_l$ e $\overline{x^j} = (x_1, \dots, x_{n_j})$

La convertibilità

La convertibilità $=_{\beta\delta\iota}$ fra termini di CIC è semplicemente definita come la chiusura riflessiva, simmetrica, transitiva e per contesti[2] delle regole di riduzione $\rhd_\beta, \rhd_\delta, \rhd_\iota$ e delle regole di unfolding.

La proprietà di normalizzazione forte assicura la decidibilità della convertibilità. Questa, a sua volta, garantisce la decidibilità del type-checking, definito qui sotto.

2.2.3 Le regole di tipaggio

Diamo ora le regole di tipaggio per i termini di CIC definendo mutuamente due giudizi. Il primo, $E[\Gamma] \vdash t : T$, asserisce che t, in un ambiente E e in un contesto Γ dipendente da E, ha tipo T. Il secondo, $\mathcal{WF}(E)[\Gamma]$, asserisce che E è un ambiente valido, ovvero contenente solo definizioni di costanti e di tipi mutuamente (co)induttivi ben tipati, e che Γ è un contesto valido in E, ovvero contenente solo dichiarazioni di variabili il cui tipo è ben tipato.

Regole del Calcolo delle Costruzioni

WF-Empty

$$\mathcal{WF}([])[[]]$$

WF-Var

$$\frac{E[\Gamma] \vdash T : s \qquad s \in \{\mathbf{Prop}, \mathbf{Set}, \mathbf{Type}(i)\} \qquad x \notin \Gamma}{\mathcal{WF}(E)[\Gamma, (x : T)]}$$

WF-Const

$$\frac{E[[]] \vdash T : s \qquad E[[]] \vdash t : T \qquad c \notin E}{\mathcal{WF}(E; c : T := t)[[]]}$$

[2] La chiusura per contesti è quella standard, che asserisce le proprietà di congruenza di $=_{\beta\delta\iota}$, e non ha relazioni con i contesti definiti in precedenza.

Ax-Prop

$$\frac{\mathcal{WF}(E)[\Gamma]}{E[\Gamma] \vdash \mathbf{Prop} : \mathbf{Type}(n)}$$

Ax-Set

$$\frac{\mathcal{WF}(E)[\Gamma]}{E[\Gamma] \vdash \mathbf{Set} : \mathbf{Type}(n)}$$

Ax-Acc

$$\frac{\mathcal{WF}(E)[\Gamma] \qquad n < m}{E[\Gamma] \vdash \mathbf{Type}(n) : \mathbf{Type}(m)}$$

Var

$$\frac{\mathcal{WF}(E)[\Gamma] \qquad (x : T) \in \Gamma}{E[\Gamma] \vdash x : T}$$

Const

$$\frac{\mathcal{WF}(E)[\Gamma] \qquad (c : T := t) \in E}{E[\Gamma] \vdash c : T}$$

Prod-SP

$$\frac{E[\Gamma] \vdash T : s_1 \quad E[\Gamma, (x : T)] \vdash U : s_2 \quad s_1 \in \{\mathbf{Prop}, \mathbf{Set}\} \quad s_2 \in \{\mathbf{Prop}, \mathbf{Set}\}}{E[\Gamma] \vdash \Pi x : T.U : s_2}$$

Prod-T

$$\frac{E[\Gamma] \vdash T : \mathbf{Type}(n_1) \quad E[\Gamma, (x : T)] \vdash U : \mathbf{Type}(n_2) \quad n_1 \leq n \quad n_2 \leq n}{E[\Gamma] \vdash \Pi x : T.U : \mathbf{Type}(n)}$$

Lam

$$\frac{E[\Gamma] \vdash \Pi x : T.U : s \quad E[\Gamma, (x : T)] \vdash t : U}{E[\Gamma] \vdash \lambda x : T.t : \Pi x : T.U}$$

App

$$\frac{E[\Gamma] \vdash t : \Pi x : U.T \quad E[\Gamma] \vdash u : U}{E[\Gamma] \vdash (t\ u) : T\{u/x\}}$$

Conv

$$\frac{E[\Gamma] \vdash T_1 : s \quad E[\Gamma] \vdash t : T_2 \quad E[\Gamma] \vdash T_1 =_{\beta\delta\iota} T_2}{E[\Gamma] \vdash t : T_1}$$

Regole per i tipi (co)induttivi

Per brevità, nelle regole che seguono, con $\star$ viene indicata la seguente dichiarazione di tipi mutuamente (co)induttivi:

$$\begin{array}{l}
\Pi x_1 : U_1 \ldots \Pi x_h : U_h. \\
\quad \{i_1 : A_1 := \{k_1^1 : C_1^1\ ;\ \ldots\ ;\ k_{n_1}^1 : C_{n_1}^1\}\ ; \\
\quad \ldots\ ; \\
\quad i_m : A_m := \{k_1^m : C_1^m\ ;\ \ldots\ ;\ k_{n_m}^m : C_{n_m}^m\} \\
\quad \}_{(CO)IND}
\end{array}$$

WF-Ind

$$\frac{\begin{array}{l}(E[[]] \vdash (\Pi x_1 : U_1 \ldots \Pi x_h : U_h.A_j : s_j)_{j=1,\ldots,m} \\ (E[[y_1 : A_1^* \ ; \ \ldots \ ; \ y_m : A_m^*]] \vdash C_l^j\{y_1/i_1 \ ; \ \ldots \ ; y_m/i_m\} : s_l^j)_{j=1,\ldots,m \ ; \ l=1,\ldots,n_j} \\ i_1, \ldots, i_m, k_1^1, \ldots, k_{n_m}^m \notin E \\ i_1, \ldots, i_m, k_1^1, \ldots, k_{n_m}^m \mathit{distinti}\end{array}}{\mathcal{WF}(E, \star)[[]]}$$

dove $A_j^* = \Pi x_1 : U_1 \ldots \Pi x_h : U_h.A_j$ per $j \in \{1, \ldots, m\}$
e con il vincolo sintattico che in $C_1^1, \ldots, C_{n_m}^m$ i primi h argomenti
delle applicazioni la cui testa è uno dei tipi induttivi $i_1, \ldots, i_m$ siano $x_1, \ldots, x_h$

Ind

$$\frac{\mathcal{WF}(E)[\Gamma] \qquad \star \in E}{E[\Gamma] \vdash i_j : \Pi x_1 : U_1 \ldots \Pi x_h : U_h.A_j} \qquad \text{se } j \leq m$$

Constr

$$\frac{\mathcal{WF}(E)[\Gamma] \qquad \star \in E}{E[\Gamma] \vdash k_l^j : \Pi x_1 : U_1 \ldots \Pi x_h : U_h.C_l^j} \qquad \begin{array}{l}\text{se} \quad j \leq m \\ \text{e} \quad l \leq n_j\end{array}$$

2.2.4 Alcune estensioni a CIC

Nella sezione precedente è stato introdotto CIC in modo che fosse al tempo stesso facilmente comprensibile e il più vicino possibile al sistema logico implementato in HELM. Quest'ultimo, però, estende la versione di CIC descritta nella sezione precedente in diverse maniere e con diversi scopi. Qui di seguito quindi vedremo due estensioni particolarmente rilevanti nell'ambito di questa tesi.

Metavariabili

Tradizionalmente, la logica tratta *termini completamente specificati* e *prove completate*. Le regole di inferenza di un sistema logico specificano che cosa *è* derivabile, e non cosa *potrebbe* esserlo sotto alcune ipotesi aggiuntive. Analogamente, le regole di tipizzazione di un sistema di tipi specificano che cosa *è* ben tipato e non cosa *potrebbe* essere derivabile facendo ulteriori ipotesi o istanziazioni. Tuttavia, i proof-assistant e i theorem-prover hanno come scopo

proprio quello di guidare l'utente nella *costruzione* di una dimostrazione, e ciò ovviamente implica che essi devono essere in grado di trattare termini e prove *incompleti*. Ad esempio, un utente potrebbe inizialmente evitare di specificare alcune ipotesi, supporre l'esistenza di un particolare termine che gode di certe proprietà e quindi procedere con la dimostrazione, accumulando "lungo la strada" dei vincoli sulla natura del termine ipotizzato ma non ancora esibito. Più tardi, quando la prova è ormai prossima ad essere completata, l'utente può esaminare i vincoli e istanziare tale termine con un valore che li soddisfi tutti. Allo stesso tempo, anche le ipotesi implicite possono essere istanziate per completare quindi la dimostrazione.

A tal fine, CIC viene esteso con l'aggiunta di *metavariabili* che corrispondono alle congetture non ancora provate. Le metavariabili, che appartengono ad una classe sintattica distinta da quella delle variabili, sono particolari termini che hanno un tipo, ma non hanno ancora un corpo. Completare la prova significa abitare il tipo di ogni metavariabile con un termine ben tipato nell'ambiente e nel contesto della metavariabile. Usando la sintassi astratta $? : T$ per indicare una metavariabile il cui tipo è T, la regola di tipaggio diventa:

Meta

$$\frac{E[\Gamma] \vdash T : s}{E[\Gamma] \vdash (? : T) : T}$$

Per esempio, se ? è una metavariabile che ha come tipo

$$\Pi A : \mathbf{Prop}.\Pi B : \mathbf{Prop}.A \to (A \to B) \to B$$

allora

$$\lambda A : Prop.\lambda B : Prop.\lambda H : A.\lambda H0 : (A \to B).(?\ A\ B\ H\ H0)$$

è un termine che ne abita il tipo.

Oltre al tipo T, ogni metavariabile ? ha associato un contesto Γ: intuitivamente, ? rappresenta quindi un termine (ancora) sconosciuto di tipo T, le cui variabili libere possono essere solo un sottoinsieme del contesto Γ.

L'insieme delle metavariabili presenti in un termine forma il suo `metasenv` (*metavariables environment*). Con questa estensione, una prova si dice non terminata se contiene almeno una metavariabile, oppure se dipende transitivamente da una costante che ne contiene.

Sulle metavariabili sono effettuabili due operazioni principali: l'*istanziazione* e la *restrizione*. L'istanziazione è l'operazione, descritta anche sopra, di dare alla metavariabile un corpo, che sia chiuso rispetto al contesto della metavariabile stessa e abbia il tipo atteso in quel contesto. La restrizione elimina alcune ipotesi dal contesto della metavariabile; è necessaria, ad esempio, durante la fase di unificazione: la prima cosa da fare per unificare due metavariabili è restringerle entrambe allo stesso contesto.

Oggetti

Definiamo *oggetti* tutte le entità che possono essere presenti in un ambiente. In CIC, esse sono le definizioni di costanti, introdotte con la regola WF-Const, e le definizioni di tipi mutuamente (co)induttivi, introdotte con la regola WF-Ind. HELM introduce, inoltre, due nuovi tipi di oggetti: le variabili e le prove.

Le variabili hanno un tipo, ma sono prive di corpo, pertanto non sono soggette a δ-riduzione e hanno la seguente regola di introduzione:

WF-DefVar

$$\frac{E[[]] \vdash T : s \qquad c \notin E}{\mathcal{WF}(E; c : T)[[]]}$$

Le prove possono essere chiuse, cioè già completate, o ancora incomplete: le prime hanno la seguente regola di *Well-Formness*:

WF-DefClosedProof

$$\frac{E[[]] \vdash T_1 : s \qquad E[[]] \vdash t : T_2 \qquad T_1 =_{\beta\delta\iota} T_2}{\mathcal{WF}(E; t : T_1)[]}$$

Per le prove incomplete abbiamo la seguente regola:

WF-DefOpenProof

$$\frac{\begin{array}{l}(\mathcal{WF}(E, m_i))_{i=1,\ldots,k} \\ E \cup \{m_1 \ldots m_k\}[] \vdash T_1 : s \\ E \cup \{m_1 \ldots m_k\}[] \vdash t : T_2 \qquad T_1 =_{\beta\delta\iota} T_2\end{array}}{\mathcal{WF}(E \cup \{m_1 \ldots m_k\}; t : T_1)[]}$$

dove gli m_i sono giudizi nella forma $d^i_1, \ldots, d^i_{k_i} \vdash ?_i : T_i$. Inoltre, per trattare le prove incomplete, dobbiamo aggiungere le regole di *Well-Formness* di un ambiente che contenga anche metavariabili:

WF-MetaDef

$$\frac{\begin{array}{l}E[d_1, \ldots, d_n] \vdash T : s \qquad s \in \{\mathbf{Prop}, \mathbf{Set}, \mathbf{Type}\} \\ (E[d_1, \ldots, d_i] \vdash T_{i+1} : s)_{i=1\ldots n} \qquad s \in \{\mathbf{Prop}, \mathbf{Set}, \mathbf{Type}\} \\ (E[d_1, \ldots, d_i] \vdash b_{i+1} : T)_{i=1\ldots n}\end{array}}{\mathcal{WF}(E, d_1, \ldots, d_n \ \vdash ?_i : T)}$$

dove con d_i sono indicate indifferentemente le definizioni e le dichiarazioni, cioè

$$\begin{array}{rcll} d_i & ::= & x_i : T_i & \text{dichiarazioni} \\ & | & x_i := b_i & \text{definizioni} \end{array}$$

2.3 Il proof-assistant Matita

Con il termine *proof-assistant* si indicano generalmente tutti quei programmi che in qualche modo "accompagnano" l'utente nella dimostrazione di un teorema, controllando e garantendo la correttezza dei singoli passi, e di conseguenza quella dell'intera procedura di prova. Tuttavia, questa definizione non dice nulla sul modo in cui sono rappresentati e manipolati gli oggetti coinvolti (teoremi e prove di essi), né su come sia effettivamente realizzata la procedura di verifica di correttezza, ed in effetti storicamente gli approcci sono stati molteplici.

Quello utilizzato da Matita può essere delineato nei seguenti punti:

- Sia i teoremi che le loro prove sono rappresentati come termini CIC ben tipati (d'ora in avanti λ-*termini*);
- Le dimostrazioni vengono costruite mediante una sequenza di applicazioni di *tattiche*, che ne rappresentano i vari passi;
- La verifica di correttezza si basa sull'isomorfismo di Curry-Howard, [GTL89] (Capitolo 3), in base al quale il teorema da dimostrare è visto come un tipo di un termine CIC, e la sua dimostrazione consiste nell'esibire un termine CIC che abbia come tipo il teorema stesso (ovvero, trovare una dimostrazione significa costruire un termine che abiti il tipo corrispondente al teorema in esame): la correttezza viene quindi stabilita da un *proof-checker* che si occupa di garantire che le dimostrazioni abbiano il tipo giusto (e che quindi è un *type-checker* per i termini CIC).

Coerentemente con il resto di HELM, ed in accordo a quanto schematizzato qui sopra, l'architettura di Matita è estremamente modulare, con un nucleo costituito dal proof-checker, intorno al quale vengono quindi assemblate le altre funzionalità del proof-assistant, come le varie tattiche a disposizione ed il proof-engine, ossia il componente che si occupa dell'applicazione delle tattiche al fine di ottenere una dimostrazione: il resto della Sezione è quindi dedicato alla descrizione di questi componenti fondamentali, tralasciando quelli, come l'interfaccia grafica o il linguaggio utilizzato per scrivere le dimostrazioni, che non sono direttamente correlati al lavoro di questa tesi.

2.3.1 Definizione di tattica

Abbiamo visto nell'Introduzione che una tattica è un procedimento che rappresenta un passo di una dimostrazione, sostituendo il goal corrente con alcuni nuovi (e "più semplici") sottogoal, a cui poi possono essere applicate altre tattiche, e così via fino ad ottenere una dimostrazione completa.

Formalmente, una tattica implementa un *ragionamento all'indietro*: se il goal corrente è la conclusione di una regola di deduzione, l'applicazione di una tattica ad esso lo rimpiazza con le sue premesse, cioè genera un nuovo sottogoal per ogni premessa, secondo uno schema che può essere parafrasato come: "se il goal G dipende (ha come premesse) $G'_1, \ldots, G'_n$, per ottenere una dimostrazione di G devo prima avere le dimostrazioni di $G'_1, \ldots, G'_n$". Per questo motivo una dimostrazione costruita con le tattiche parte dalla tesi e non si può dire che sia terminata finché ci sono goal da ridurre: un goal viene chiuso quando l'applicazione di una tattica a quel goal non genera sottogoal.

Ancora più formalmente, una tattica può essere definita nel modo seguente, in accordo con [Sac04]:

Definizione 2.4 (Tattica). Definiamo *tattica* una funzione che, dato un goal G (ovvero un contesto locale più un tipo da abitare) che soddisfa una lista di assunzioni (ipotesi) P, restituisce una coppia (L, t) dove $L = L_1, \ldots, L_n$ è una lista, eventualmente vuota, di sottogoal e t è una funzione che, data una lista $l = l_1, \ldots, l_n$ di termini tali che l_i abita L_i per ogni $i \in 1, \ldots, n$, restituisce un termine $t(l)$ che abita G.

Il λ-termine che, alla fine della dimostrazione, abiterà la tesi viene costruito passo passo dalle tattiche: questa definizione sottolinea il fatto che una tattica, oltre a modificare lo stato corrente della dimostrazione, ha anche il compito di generare il "pezzetto" di λ-termine di sua competenza, ovvero la funzione t che trasforma i λ-termini che abitano i sottogoal nel λ-termine che abita il goal a cui la tattica è stata applicata.

2.3.2 Il proof-checker

In HELM tutti i giudizi logici sono di tipaggio, pertanto il vero cuore del sistema è il proof-checker (o type-checker), cioè la componente software che verifica

la correttezza delle prove, ovvero, secondo l'isomorfismo di Curry-Howard, il corretto tipaggio dei termini e, ancora, l'aderenza dei programmi alle specifiche. L'architettura di HELM è stata studiata in modo che la sua "correttezza" sia garantita esclusivamente da quella del type-checker. In altre parole, se CIC è consistente e il type-checker non contiene bug, allora ogni prova sarà effettivamente un termine che dimostra la tesi, ovvero che abita in CIC il termine che la rappresenta. Come conseguenza, ogni possibile estensione al sistema che non interessi il type-checker è sicura: in particolare, la definizione di nuove tattiche è un task non critico per la correttezza dell'intero sistema.

Una descrizione dell'implementazione del proof-checker esula dagli obiettivi di questa tesi, e pertanto si rimanda a [Sac04] per gli approfondimenti. Tuttavia, possiamo dire che essa è piuttosto standard nell'ambito dell'implementazione di type-checker per λ-calcoli basati sulla rappresentazione delle variabili con indici di DeBrujin, e segue abbastanza naturalmente dalle regole di tipaggio di CIC introdotte nella Sezione precedente.

2.3.3 Il proof-engine

Come accennato sopra, il proof-engine è il componente che si occupa di mantenere lo stato corrente della dimostrazione, applicare le tattiche ed effettuare le operazioni di backtracking in caso di fallimento durante l'esecuzione.

Una dimostrazione non terminata è codificata in HELM con una struttura dati complessa mostrata schematicamente in figura 2.2. Una prova (`proof`) è formata da un `uri` che la identifica[3], una lista di sottogoal aperti (`metasenv`, ricordiamo che una prova aperta è una metavariabile e che questa può comparire in un ambiente), eventualmente vuota, la prova in forma di λ-termine, incompleto, cioè contenente delle metavariabili, se la lista di sottogoal aperti non è vuota, e la tesi che la prova dimostra, anch'essa in forma di λ-termine. Ogni sottogoal è identificato da un numero e contiene un contesto, cioè una lista di definizioni e dichiarazioni, e il tipo del goal che dimostra.

La struttura dati `proof` accoppiata con il numero del goal correntemente

[3] Ai fini della presente tesi, non è rilevante sapere come effettivamente tale uri sia composto, ma si può semplicemente assimilare ad un identificativo univoco.

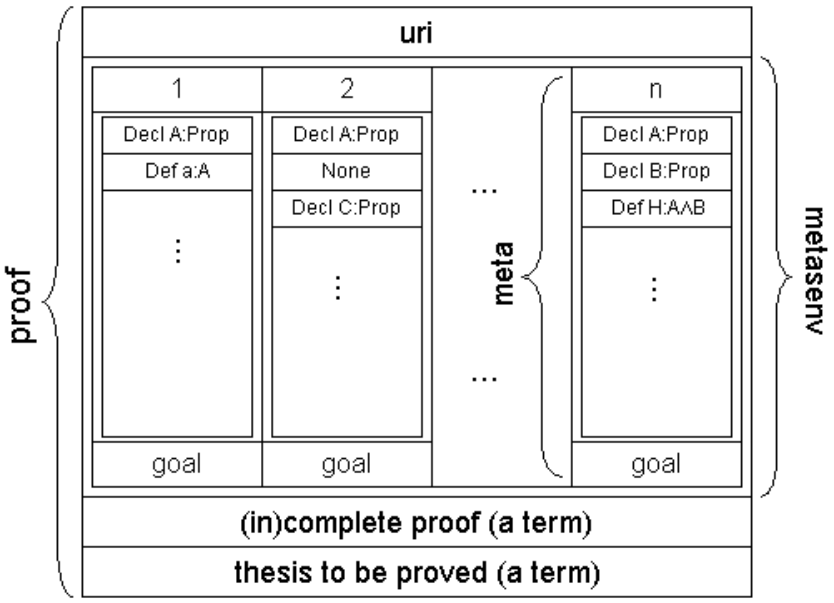

Figura 2.2: Struttura dati della prova

visualizzato nell'interfaccia forma lo `status` della dimostrazione; dal momento che l'esecuzione di una tattica può chiudere il goal senza aprirne di nuovi, o aprendone più di uno, questa struttura non basta più a identificare il risultato di questa esecuzione, quindi, dal punto di vista implementativo, una tattica è una funzione che trasforma uno `status` in una coppia che contiene la nuova `proof` e la lista dei goal che la tattica ha aperto, oppure solleva l'eccezione `Fail`, in caso di fallimento.

2.3.4 La tattica `auto`

Concludiamo il capitolo con una descrizione generale del funzionamento della tattica `auto` attualmente disponibile in Matita. Anche in questo caso, rimandiamo a lavori più specifici (ed in particolare [Sel04]) per eventuali approfondimenti.

`Auto` è stata pensata per ottenere la risoluzione dei goal solamente grazie all'utilizzo dei teoremi presenti in libreria e delle ipotesi nel contesto. In pratica, i goal di cui si vuole ottenere la dimostrazione tramite `auto` sono quelli soddisfacibili con una seri di applicazioni successive di teoremi. `Auto` quindi non proverà ad utilizzare altre tattiche, eccetto la tattica elementare `apply` che

applica un teorema al goal.

L'idea di base dell'algoritmo che implementa la tattica è abbastanza semplice: finché la lista dei goal aperti non è vuota, si seleziona un goal da essa, si cercano nella libreria (e nel contesto del goal) i teoremi (e le ipotesi) applicabili, quindi si sostituisce il goal con un insieme di liste di sottogoal, ognuna delle quali è il risultato dell'applicazione di uno dei teoremi (o delle ipotesi) trovati al passo precedente; il procedimento viene quindi ripetuto finché non si ottiene un insieme in cui almeno una delle liste di sottogoal è vuota, oppure non ci sono più teoremi applicabili.

Questa idea di base viene quindi raffinata in diversi modi, al fine di migliorare l'efficienza della tattica. Tali ottimizzazioni comprendono l'utilizzo di limiti di profondità e di larghezza per l'albero "AND-OR" formato dai sottogoal aperti, una strategia di ordinamento dei sottogoal generati in modo da selezionare quelli che più probabilmente porteranno ad una soluzione, il filtraggio della lista di teoremi applicabili per eliminare quelli ridondanti, ed altre ancora [Sel04]. Esse hanno consentito di ottenere un notevole incremento nelle prestazioni, ma tuttavia non hanno eliminato il principale problema di efficienza di `auto`, ovvero il trattamento del predicato di uguaglianza. Nel prossimo Capitolo verranno spiegati i motivi per cui esso risulta così problematico, e verrà illustrata una tecnica rivelatasi particolarmente efficace per la soluzione[4] di questo problema, ovvero la paramodulazione.

[4]intesa come sensibile miglioramento delle prestazioni.

Capitolo 3

La paramodulazione nella logica del primo ordine

Questo Capitolo espone la teoria della paramodulazione nell'ambito in cui essa è stata sviluppata ed è principalmente adottata, cioè il theorem proving basato sul metodo di risoluzione per la logica del primo ordine [Rob65, BG01]. I risultati qui esposti sono quindi stati adattati a CIC[1] ed utilizzati per la progettazione della nuova tattica `auto paramodulation`, la cui descrizione è oggetto dei Capitoli seguenti. L'esposizione segue [NR01], in cui si possono trovare le dimostrazioni dei risultati qui solo enunciati e tutti i dettagli qui omessi.

Nel resto del Capitolo, si assume che il metodo di risoluzione per la logica del primo ordine sia noto. Per eventuali approfondimenti su di esso, si rimanda a [BG01].

3.1 Introduzione e nozioni preliminari

3.1.1 Problemi del predicato di uguaglianza

Alla fine del Capitolo precedente abbiamo accennato al fatto che il problema principale di `auto` è dato dall'inefficienza nel trattamento dei predicati di uguaglianza (=). Il problema è esattamente lo stesso che si presenta anche nel calcolo dei predicati, ed è dovuto alla definizione di = tramite l'insieme di assiomi mostrato in Figura 3.1.

[1] in effetti, come vedremo in seguito, ad un suo sottoinsieme.

	$\rightarrow$	$x = x$	(riflessività)
$x = y$	$\rightarrow$	$y = x$	(simmetria)
$x = y \wedge y = z$	$\rightarrow$	$x = z$	(transitività)
$x_1 = y_1 \wedge \ldots \wedge x_n = y_n$	$\rightarrow$	$f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)$	(monotonicità)

Figura 3.1: Assiomi che definiscono =

È noto infatti che l'applicazione di questi assiomi nella dimostrazione di un goal, ed in particolare della transitività, causa molto velocemente un'esplosione del numero di nuovi sottogoal aperti.

Esempio 3.1. *Supponiamo di voler dimostrare il goal*

$$(app\ (app\ (app\ S\ K)\ K)\ z) = z$$

nel contesto:

A : SET
app : $A \rightarrow (A \rightarrow A)$
S : A
K : A
H : $\forall x, y, z : A.(app\ (app\ (app\ S\ x)\ y)\ z) = (app\ (app\ x\ z)\ (app\ y\ z))$
$H1$: $\forall x, y : A.(app\ (app\ K\ x)\ y) = x$
$H2$: $\forall x, y : A.(app\ (app\ K\ y)\ (app\ S\ x)) = x.$

Possiamo ottenere una dimostrazione per transitività in questo modo:

- *Applichiamo la transitività al goal, ottenendo due nuovi sottogoal*

$$(app\ (app\ (app\ S\ K)\ K)\ z) = ?1 \quad e \quad ?1 = z;$$

- *Applicando H a K, K e z, possiamo dimostrare il primo sottogoal, istanziando quindi $?1$ con*

$$(app\ (app\ K\ z)\ (app\ K\ z));$$

- *Applicando la sostituzione $\{?1 \mapsto (app\ (app\ K\ z)\ (app\ K\ z))\}$, il secondo sottogoal diventa*

$$(app\ (app\ K\ z)\ (app\ K\ z)) = z,$$

che può essere dimostrato applicando $H1$ a z e $(app\ K\ z)$.

Tuttavia, la ricerca della dimostrazione avrebbe potuto procedere diversamente: ad esempio, avremmo potuto dimostrare il secondo sottogoal usando $H2$ *applicata a* z *e ad una nuova metavariabile* ?2*, istanziando* ?1 *con*

$$(app\ (app\ K\ ?2)\ (app\ S\ z)).$$

A questo punto si sarebbe potuto procedere cercando di dimostrare il primo sottogoal, diventato ora

$$(app\ (app\ (app\ S\ K)\ K)\ z) = (app\ (app\ K\ ?2)\ (app\ S\ z)),$$

solo per accorgersi (magari dopo alcuni tentativi di applicazione di simmetria, riflessività, monotonicità o altre transitività) che esso non può essere dimostrato.

L'idea del metodo basato su paramodulazione è quella di trattare l'uguaglianza in modo speciale, considerandola come un costrutto del linguaggio invece che un predicato come gli altri: in questo modo, un'equazione $a = b$ è vista come una *regola di riscrittura* che indica che posso riscrivere un termine $T(a)$ (cioè che contiene a) in $T(b)$ (o viceversa).

Esempio 3.2. *Considerando l'esempio precedente, se trattiamo* H*,* $H1$ *e* $H2$ *come regole di riscrittura* [2]*:*

$$\begin{aligned} H: &\quad \forall x,y,z:A.(app\ (app\ (app\ S\ x)\ y)\ z) \Rightarrow (app\ (app\ x\ z)\ (app\ y\ z)) \\ H1: &\quad \forall x,y:A.(app\ (app\ K\ x)\ y) \Rightarrow x \\ H2: &\quad \forall x,y:A.(app\ (app\ K\ y)\ (app\ S\ x)) \Rightarrow x, \end{aligned}$$

l'unica strada possibile per dimostrare il goal è:

- *utilizzando* H*, riscrivere il goal in*

$$(app\ (app\ K\ z)\ (app\ K\ z)) = z;$$

- *utilizzando* $H1$*, riscrivere il nuovo goal in*

$$z = z;$$

[2] qui $\Rightarrow$ indica il verso della riscrittura stessa

- *a questo punto, concludere per riflessività.*

Prima di procedere con la descrizione formale di tale approccio, è opportuno però ricordare alcune nozioni, peraltro assolutamente standard, che definiscono precisamente il dominio in cui stiamo operando.

3.1.2 Termini, sostituzioni, riscritture

Definizione 3.1 (Termini del primo ordine). Siano $\mathcal{F}$ un insieme di simboli di funzione (di arietà diversa), e $\mathcal{X}$ un insieme di simboli di variabili. Si definisce l'insieme di *termini del primo ordine* su $\mathcal{F}$ e $\mathcal{X}$, indicato con $\mathcal{T}(\mathcal{F}, \mathcal{X})$, il più piccolo insieme contenente $\mathcal{X}$ tale che $f(x_1, \ldots, x_n) \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ se $f \in \mathcal{F}$, f ha arietà n e $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F}, \mathcal{X})$. Il sottoinsieme di $\mathcal{T}(\mathcal{F}, \mathcal{X})$ contenente solo i termini senza variabili (*ground*) è indicato con $\mathcal{T}(\mathcal{F})$.

Definizione 3.2 (Sostituzione). Una *sostituzione* σ è una funzione dalle variabili ai termini. Normalmente si usa la notazione postfissa per indicare l'applicazione di una sostituzione, ovvero l'applicazione di σ ad x si indica con $x\sigma$. Una sostituzione può essere estesa ad una funzione dai termini ai termini nel seguente modo: se t è un termine, $t\sigma$ è il termine che si ottiene sostituendo ogni variabile x di t con $x\sigma$.

Si dice che un termine t_1 *fa match* con un termine t_2 se esiste una sostituzione σ tale che $t_1 = t_2\sigma$ (ed in questo caso t_1 si dice *istanza* di t_2).

Due termini t_1 e t_2 si dicono *unificabili* se esiste una sostituzione σ (detta *unificatore*) tale che $t_1\sigma = t_2\sigma$. Se ogni altro unificatore θ è una particolare istanza di σ, ovvero se si ha che $t_1\theta = t_1\sigma\sigma' = t_2\theta = t_2\sigma\sigma'$ per una qualche σ', allora σ si dice *unificatore più generale* di t_1 e t_2, e si indica con $mgu(t_1, t_2)$.

Definizione 3.3 (Posizione). Una *posizione* è o una sequenza vuota Λ oppure una sequenza $p.i$, dove p è una posizione e i un naturale. Le nozioni di *posizione in un termine* e *sottotermine di t in posizione p*, indicato con $t|_p$, sono definite come:

- Λ è una posizione in t e $t|_\Lambda \equiv t$;

- Se $t|_p \equiv f(t_1, \ldots, t_n)$, con $n > 0$, allora $p.1, \ldots, p.n$ sono posizioni in t e $t|_{p.i} \equiv t_i$ per ogni $i \in \{1, \ldots, n\}$.

Definizione 3.4 (Riscrittura). Una *regola di riscrittura* è una coppia ordinata di termini (t_1, t_2), e si indica con $t_1 \Rightarrow t_2$. Un *sistema di riscrittura* R è un insieme di regole di riscrittura. Si dice che un termine t_1 *si riscrive in* t_2 con R se $t_1 \rightarrow_R t_2$, dove $\rightarrow_R$ indica la più piccola relazione monotona tale che $l\sigma \rightarrow_R r\sigma$ per ogni $l \Rightarrow r \in R$ ed ogni σ. Una relazione $\rightarrow$ si dice *monotona* se $t_1 \rightarrow t_2$ implica $u[t_1]_p \rightarrow u[t_2]_p$, dove $u[t]_p$ indica un termine u il cui sottotermine in posizione p è t.

3.1.3 Ordinamenti tra termini

Per poter trattare le equazioni tra termini come regole di riscrittura, alla luce della Definizione 3.4, è necessario stabilire un qualche tipo di ordinamento tra i due termini a destra e a sinistra dell'=. Ciò può essere fatto in maniera banale semplicemente mappando ogni equazione $t_1 = t_2$ in una coppia di regole $t_1 \Rightarrow t_2$ e $t_2 \Rightarrow t_1$. Tuttavia questo approccio porterebbe ad una grande inefficienza, in quanto permetterebbe la generazione di un elevato numero di passi di inferenza ridondanti.

Esempio 3.3. *Riferendoci sempre all'Esempio 3.1, questo tipo di soluzione al problema dell'ordinamento porterebbe ad avere le seguenti regole di riscrittura:*

H : $\forall x, y, z : A.(app\ (app\ (app\ S\ x)\ y)\ z) \Rightarrow (app\ (app\ x\ z)\ (app\ y\ z))$
H' : $\forall x, y, z : A.(app\ (app\ x\ z)\ (app\ y\ z)) \Rightarrow (app\ (app\ (app\ S\ x)\ y)\ z)$
$H1$: $\forall x, y : A.(app\ (app\ K\ x)\ y) \Rightarrow x$
$H1'$: $\forall x, y : A.x \Rightarrow (app\ (app\ K\ x)\ y)$
$H2$: $\forall x, y : A.(app\ (app\ K\ y)\ (app\ S\ x)) \Rightarrow x$
$H2'$: $\forall x, y : A.x \Rightarrow (app\ (app\ K\ y)\ (app\ S\ x))$.

È chiaro che con l'introduzione di queste nuove regole di riscrittura la procedura di costruzione della dimostrazione descritta all'Esempio 3.2 non è più l'unica possibile, in quanto ad ogni passo è possibile applicare la riscrittura "inversa", potenzialmente andando in loop.

In effetti, per ovviare a questo problema può essere imposto un ordinamento tra i termini, in modo che le inferenze possibili siano solo quelle che riscrivono un termine t_1 in t_2 in modo che t_2 non sia maggiore di t_1 rispetto ad una relazione d'ordine parziale $\succ$, che deve essere un *ordinamento di riduzione*:

Definizione 3.5 (Ordinamento di riduzione). Un ordinamento $\succ$ su $\mathcal{T}(\mathcal{F}, \mathcal{X})$ è un *ordinamento di riduzione* se soddisfa le seguenti ipotesi:

1. è *stabile rispetto alle sostituzioni*: $t_1 \succ t_2$ implica $t_1\sigma \succ t_2\sigma$ per ogni sostituzione σ;
2. tutti i simboli di funzione in $\mathcal{F}$ sono *monotoni* rispetto a $\succ$: $t_1 \succ t_2$ implica $f(u_1, \ldots, u_{i-1}, t_1, u_{i+1}, \ldots, u_n) \succ f(u_1, \ldots, u_{i-1}, t_2, u_{i+1}, \ldots, u_n)$ per ogni i;
3. $\succ$ è *ben fondato*, ovvero non esiste nessuna catena infinita $t_1 \succ t_2 \succ \ldots$;
4. è totale sui termini ground.

Una regola di paramodulazione che tiene conto dell'ordinamento tra termini prende il nome di *superposizione*.

3.2 Il calcolo di base

Possiamo ora introdurre le regole di inferenza per il calcolo di paramodulazione per la logica del primo ordine. In quanto segue, ci concentreremo solamente su sistemi puramente equazionali, ovvero in cui sia il goal da dimostrare che le ipotesi ed i teoremi applicabili sono costituiti da *clausole* formate da una sola equazione. La ragione di ciò è che nell'ambito di questa tesi è stato preso in considerazione solamente questo tipo di scenario. Tuttavia, i risultati qui esposti possono essere generalizzati a clausole di tipo qualsiasi. Il lettore interessato può fare riferimento a [NR01].

3.2.1 Regole di inferenza

In quanto segue, un'equazione $t_1 = t_2$ verrà indicata con $t_1 = t_2 \rightarrow$ se si tratta di un goal, e con $\rightarrow t_1 = t_2$ se invece si tratta di un teorema (o ipotesi). Questa

notazione deriva da quella più generale per le clausole $\Gamma \rightarrow \Delta$ nell'ambito della dimostrazione per risoluzione, in cui Γ è l'insieme di premesse e Δ l'insieme di conclusioni: un teorema (o ipotesi) è quindi una clausola senza premesse, mentre un goal è una clausola che mi permette di ottenere una contraddizione. Sempre in accordo con la terminologia usata nel theorem proving per risoluzione, talvolta i goal verranno chiamati *clausole negative*, ed i teoremi *clausole positive*.

Inoltre, con $t|_p$ si denota il sottotermine di t in posizione p, e con $u[t]_p$ il termine ottenuto sostituendo $u|_p$ con t.

Nel calcolo di base, nel caso di un sistema puramente equazionale le uniche regole di inferenza sono la superposizione, che stabilisce in che modo è possibile riscrivere un'equazione in un'altra "più piccola"[3] usando i teoremi e le ipotesi come regole di riscrittura, e la regola di risoluzione dell'uguaglianza, che permette di concludere la dimostrazione quando i due lati del goal sono unificabili. La prima regola è presente in due varianti, sinistra e destra, a seconda che l'equazione da riscrivere sia un goal oppure no:

superposizione sinistra

$$\frac{\rightarrow l = r \qquad t_1 = t_2 \rightarrow}{(t_1[r]_p = t_2 \rightarrow)\sigma}$$

se $\sigma = mgu(l, t_1|_p)$, $t_1|_p$ non è una variabile, $l\sigma \not\preceq r\sigma$ e $t_1\sigma \not\preceq t_2\sigma$;

superposizione destra

$$\frac{\rightarrow l = r \qquad \rightarrow t_1 = t_2}{(\rightarrow t_1[r]_p = t_2)\sigma}$$

se $\sigma = mgu(l, t_1|_p)$, $t_1|_p$ non è una variabile, $l\sigma \not\preceq r\sigma$ e $t_1\sigma \not\preceq t_2\sigma$;

risoluzione di uguaglianza

$$\frac{t_1 = t_2 \rightarrow}{\square}$$

se esiste $\sigma = mgu(t_1, t_2)$.

Teorema 3.6 (Completezza). *Il sistema di inferenza appena descritto è completo per refutazione (per clausole puramente equazionali), cioè se $\mathcal{H}$ è un'insieme di ipotesi e $t_1 = t_2$ è un goal da dimostrare, se $t_1 = t_2$ è vero sotto le ipotesi*

[3] o meglio, "non più grande"

$\mathcal{H}$ allora l'applicazione ripetuta delle regole di superposizione e di risoluzione di uguaglianza all'insieme $\mathcal{H} \cup \{t_1 = t_2 \rightarrow\}$ genera la clausola vuota $\square$.

Esempio 3.4. *Supponiamo di voler dimostrare il goal $f(d) = d$ avendo a disposizione le ipotesi $c = d$ e $f(c) = d$. Supponiamo inoltre che l'ordinamento di riduzione $\succ$ sia tale che $f \succ c \succ d$. La tabella seguente mostra come come è possibile dimostrare il goal mediante paramodulazione.*

Insieme di clausole	*Regola applicata*
$\rightarrow \quad c = d$ $\rightarrow \quad f(d) = d$ $f(c) = d \quad \rightarrow$	*superposizione sinistra su $f(c) = d$, usando $c \Rightarrow d$*
$\rightarrow \quad c = d$ $\rightarrow \quad f(d) = d$ $f(c) = d \quad \rightarrow$ $f(d) = d \quad \rightarrow$	*superposizione sinistra su $f(d) = d$, usando $f(d) \Rightarrow d$*
$\rightarrow \quad c = d$ $\rightarrow \quad f(d) = d$ $f(c) = d \quad \rightarrow$ $f(d) = d \quad \rightarrow$ $d = d \quad \rightarrow$	*risoluzione di uguaglianza su $d = d$*
$\rightarrow \quad c = d$ $\rightarrow \quad f(d) = d$ $f(c) = d \quad \rightarrow$ $f(d) = d \quad \rightarrow$ $\square$	

3.2.2 Criteri di ordinamento

Dal punto di vista teorico, la generica relazione d'ordine tra termini $\succ$ può essere istanziata con qualsiasi tipo di ordinamento che soddisfi la Definizione 3.5. Tuttavia nei sistemi reali la scelta ricade quasi sempre su uno tra i seguenti criteri (o varianti di essi): *lexicographic path ordering* (LPO), *recursive path ordering* (RPO) e *Knuth-Bendix ordering* (KBO), quest'ultimo anche nella variante detta *non ricorsiva*.

Tutti questi criteri si basano su un ordinamento ausiliario, ben fondato e totale, $\succ_{\mathcal{F}}$, chiamato *precedenza*.

Path ordering (LPO ed RPO)

Definizione 3.7. Sia $\succ$ un ordinamento di riduzione. L'*estensione lessicografica di* $\succ$ è la relazione $\succ^{lex}$ tra tuple di termini definita come:

$$\langle t_1, \ldots, t_n \rangle \succ^{lex} \langle u_1, \ldots, u_n \rangle \quad \text{se} \quad t_1 = u_1, \ldots t_{k-1} = u_{k-1} \text{ e } t_k \succ u_k$$

per qualche k in $1 \ldots n$. Se $\succ$ è ben fondato, anche $\succ^{lex}$ lo è.

Definizione 3.8. Un *multi insieme* di termini è una funzione M da un insieme di termini S ad $\mathbb{N}$. Un termine t appartiene ad M, $t \in M$, se $M(x) > 0$. L'unione di due multi insiemi, M ed N, è definita come $(M \cup N)(x) = M(x) + N(x)$, l'intersezione $(M \cap N)(x) = min(M(x), N(x))$. M è vuoto ($\emptyset$) se $M(x) = 0$ per ogni x in S. La relazione di uguaglianza $==$ sui multi insiemi è la più piccola relazione tale che $\emptyset == \emptyset$ e

$$S \cup t == S' \cup u \text{ se } t = u \wedge S == S'.$$

L'*estensione ai multi insiemi di* $\succ$ è il più piccolo ordinamento $\succ^{mul}$ sui multi insiemi di termini tale che

$$M \cup \{t\} \succ^{mul} N \cup \{u_1, \ldots, u_n\} \text{ se } M == N \text{ e } t \succ u_i \text{ per ogni } i \in 1 \ldots n.$$

Supponiamo che l'insieme $\mathcal{F}$ di simboli di funzione sia l'unione di due insiemi disgiunti di simboli, *lex* e *mul*, il primo composto dai simboli aventi *status* "lessicografico", il secondo da quelli aventi status "multi insieme". Inoltre, indichiamo con $=_{mul}$ l'uguaglianza tra termini ground modulo la permutazione degli argomenti delle funzioni aventi status "multi insieme", ovvero $f(t_1, \ldots, t_n) = g(u_1, \ldots, u_n)$ se $f = g$ e $t_{\pi(i)} =_{mul} u_i$ per $1 \leq i \leq n$, dove π è una permutazione di $1 \ldots n$ (che è l'identità se $f \in lex$). Sotto queste ipotesi, possiamo definire RPO e LPO come segue:

Definizione 3.9 (Recursive path ordering (RPO)). $t \succ_{rpo} x$ se x è una variabile che è un sottotermine proprio di t, oppure $t \equiv f(t_1, \ldots, t_n) \succ_{rpo} u \equiv g(u_1, \ldots, u_m)$ se è soddisfatta almeno una delle seguenti condizioni:

- $t_i \succ_{rpo} u$ oppure $t_i =_{mul} u$ per qualche $i \in \{1, \ldots, n\}$;
- $f \succ_{\mathcal{F}} g$ e $t \succ_{rpo} u_j$ per ogni $j \in \{1, \ldots, m\}$;
- $f \equiv g$ e $f \in mul$ e $\{t_1, \ldots, t_n\} \succ_{rpo}^{mul} \{u_1, \ldots, u_n\}$;
- $f \equiv g$ e $f \in lex$, $\langle t_1, \ldots, t_n \rangle \succ_{rpo}^{lex} \langle u_1, \ldots, u_n \rangle$ e $t \succ_{rpo} u_j$ per ogni $j \in \{1, \ldots, n\}$,

in cui $\succ_{rpo}^{lex}$ e $\succ_{rpo}^{mul}$ sono l'estensione lessicografica e ai multi insiemi rispettivamente di $\succ_{rpo}$.

Definizione 3.10 (Lexicographic path ordering (LPO)). L'ordinamento LPO è un caso particolare di RPO, in cui $\mathcal{F} = lex$, ovvero tutti i simboli di funzione hanno status "lessicografico".

Ordinamenti di Knuth-Bendix (KBO e KBO non ricorsivo)[4]

Definizione 3.11 (Peso dei termini). Sia w una funzione da $\mathcal{F}$ in $\mathbb{N}$, tale che $w(f) > 0$ per ogni $f \in \mathcal{F}$. Sia inoltre $C_u(t)$ il numero di occorrenze del termine u come sottotermine di t. Il *peso* di un termine t, indicato con $|t|$, è un polinomio a coefficienti naturali definito sull'insieme delle variabili $\mathcal{X}$ nel seguente modo:

$$|t| = \sum_{f \in \mathcal{F}} C_f(t) \cdot w(f) + \sum_{x \in \mathcal{X}} C_x(t) \cdot x.$$

È importante notare che il peso di un termine ground è sempre un naturale.

Sui pesi dei termini sono definite due relazioni d'ordine parziale, $>$ e $\succcurlyeq$, nel seguente modo:

Definizione 3.12 (Relazioni tra pesi). Siano $|t_1| = \alpha_0 + \alpha_1 x_1 + \cdots + \alpha_n x_n$ e $|t_2| = \beta_0 + \beta_1 x_1 + \cdots + \beta_n x_n$ due pesi di termini, in cui alcuni degli α_i e β_i possono essere zero. Allora $|t_1| > |t_2|$ se $\alpha_i - \beta_i \geq 0$ per ogni i e almeno per un i vale $\alpha_i - \beta_i > 0$. Si ha invece che $|t_1| \succcurlyeq |t_2|$ se $\alpha_i - \beta_i \geq 0$ per ogni $i > 0$ e $(\alpha_1 - \beta_1) + \cdots + (\alpha_n - \beta_n) \geq -(\alpha_0 - \beta_0)$.

A questo punto possiamo definire gli ordinamenti KBO nelle due varianti:

[4]Il contenuto di questa sezione è una semplificazione di quanto esposto in [Ria03].

Definizione 3.13 (Ordinamento di Knuth-Bendix (KBO)). $t \succ_{kbo} u$ se e solo se è soddisfatta almeno una delle condizioni seguenti:

- $|t| > |u|$;
- $|t| \succcurlyeq |u|$, $t \equiv f(\dots)$, $u \equiv g(\dots)$ e $f \succ_{\mathcal{F}} g$;
- $|t| \succcurlyeq |u|$, $t \equiv f(t_1, \dots, t_n)$, $u \equiv f(u_1, \dots, u_n)$ e per qualche $i \in \{1, \dots, n\}$ si ha che $t_1 = u_1, \dots, t_{i-1} = u_{i-1}, t_i \succ_{kbo} u_i$.

Definizione 3.14 (KBO non ricorsivo). $t \succ_{nr-kbo} u$ se e solo se è soddisfatta almeno una delle condizioni seguenti:

- $|t| > |u|$;
- $|t| \succcurlyeq |u|$ e $t \gg u$,

in cui $t \gg u$ è definito in questo modo:

1. Se almeno uno tra t ed u è una variabile, $t \gg u$ non vale (i due termini sono *inconfrontabili*);
2. Se $t \equiv f(t_1, \dots, t_n)$ e $u \equiv g(u_1, \dots, u_n)$, $t \gg u$ se:
   - $f \succ_{\mathcal{F}} g$, oppure
   - $f \equiv g$ e, per qualche i, $t_1 = u_1, \dots, t_{i-1} = u_{i-1}$ e $t_i \gg u_i$.

La differenza tra le due varianti sta nel fatto che la definizione della seconda non fa riferimento a $\succ_{nr-kbo}$ al suo interno: in questo senso possiamo quindi parlare di variante *non ricorsiva*, nonostante $\gg$ sia ricorsivamente definito. La differenza, oltre che dal punto di vista teorico (le due definizioni producono infatti ordinamenti differenti, e nessuno dei due è una generalizzazione dell'altro), ha un'importanza anche dal punto di vista implementativo: la variante non ricorsiva, infatti, è computazionalmente meno costosa da implementare, ed è quindi talvolta preferibile (si veda [Ria03] per i dettagli).

3.2.3 Selezione delle inferenze

La procedura di dimostrazione per risoluzione con paramodulazione consiste nel dimostrare l'insoddisfacibilità dell'insieme di clausole formato dalle ipotesi e dal goal, mediante l'applicazione successiva delle regole di inferenza fino ad ottenere un insieme contenente la clausola vuota $\Box$[5]. In generale, l'ordine in cui applicare le regole di inferenza non è univoco, in quanto ad ogni istante ci possono essere diverse scelte di regole applicabili a diverse equazioni. Dal punto di vista della completezza del metodo, le diverse strategie di *selezione delle inferenze* sono equivalenti, a patto che soddisfino il requisito di essere *fair*:

Definizione 3.15 (Strategia di selezione fair). Una strategia di selezione delle inferenze si dice *fair* se ogni inferenza applicabile in un certo stato della procedura di risoluzione o viene applicata ad un certo punto, oppure diventa inapplicabile a seguito di qualche semplificazione.

Tuttavia, da un punto di vista pratico ci sono enormi differenze tra strategie diverse: come verrà illustrato al Capitolo 4, una buona strategia di selezione è cruciale per l'efficienza della procedura. Un aiuto nella ricerca di una strategia efficace ci viene dalla teoria, con un risultato che dice che una strategia che possiamo definire "greedy", ovvero che consiste nel cercare di applicare ad ogni passo un'inferenza che coinvolga il goal (cioè superposizione sinistra o risoluzione di uguaglianza), ed applicare la superposizione destra solo quando ciò non è possibile, preserva la completezza della procedura [NR01].

3.3 Ridondanza e saturazione

Nella nostra esposizione, fino ad ora abbiamo descritto una procedura che ad ogni passo *aggiunge* nuove clausole, ottenute applicando le regole di inferenza, all'insieme di partenza, fino ad ottenere la generazione della clausola vuota. Una procedura di questo tipo, però, ha come effetto la generazione di un elevato numero di *clausole ridondanti*, come tautologie o casi particolari di istanze più generali.

[5] essendo, come è noto, il problema semidecidibile, non è detto che il processo termini.

La presenza di tali clausole ridondanti ha evidentemente un grande impatto negativo sull'efficienza della procedura, e sarebbe quindi auspicabile avere un metodo per l'eliminazione della ridondanza, mediante l'utilizzo di *regole di eliminazione e semplificazione*. L'idea è quella applicare queste regole dopo ogni passo di inferenza, prima per semplificare le nuove clausole generate, e quindi per semplificare le clausole già presenti alla luce di quelle appena generate. Convenzionalmente, la prima operazione è detta semplificazione *in avanti*, mentre la seconda *all'indietro*.

Esempio 3.5. *Per illustrare il funzionamento delle semplificazioni in avanti e all'indietro, consideriamo il seguente insieme di equazioni:*

$$\begin{aligned} 1.\quad f(a,x) &= x \\ 2.\quad f(x,a) &= f(x,b) \end{aligned}$$

Usando come ordinamento di riduzione LPO con precedenza $f \succ_{\mathcal{F}} a \succ_{\mathcal{F}} b$*, possiamo applicare una superposizione ottenendo l'equazione*

$$f(a,b) = a,$$

alla quale può essere applicata la semplificazione in avanti: utilizzando l'equazione 1, infatti, essa può essere riscritta in

$$b = a$$

A questo punto, l'insieme di equazioni diventa:

$$\begin{aligned} 1.\quad f(a,x) &= x \\ 2.\quad f(x,a) &= f(x,b) \\ 3.\quad a &= b \end{aligned}$$

A questo punto, quindi, possiamo applicare la semplificazione all'indietro, riscrivendo grazie alla nuova equazione 3 l'equazione 2 come $f(x,b) = f(x,b)$*. La rimozione di questa tautologia è un altro passo di semplificazione all'indietro. Infine, anche l'equazione 1 può essere semplificata usando 3, ottenendo*

$$f(b,x) = x.$$

Pertanto, l'insieme risultante dopo le semplificazioni è:

$$\begin{aligned} a &= b \\ f(b,x) &= x. \end{aligned}$$

Questo tipo di strategia di risoluzione, che oltre alle regole di inferenza comprende anche regole di semplificazione e di eliminazione di clausole ridondanti, prende il nome di *saturazione*. La procedura, infatti, porta idealmente alla costruzione di un insieme di clausole *saturato*, cioè chiuso rispetto al sistema di inferenza *modulo inferenze ridondanti*. Nei sistemi reali, in effetti, non sempre tutta la ridondanza è eliminata, in quanto ciò a volte può risultare eccessivamente costoso. L'argomento verrà ripreso nel Capitolo 4, in cui viene discussa la nostra implementazione. La prossima Sezione, invece, è dedicata alla presentazione di alcune regole di semplificazione compatibili con la procedura di risoluzione, il cui utilizzo cioè preserva la completezza del metodo.

3.3.1 Regole di semplificazione

La sintassi utilizzata per la descrizione delle regole di semplificazione è la seguente:

$$S \rightarrow S',$$

dove S è l'insieme di clausole attuali, e S' è il risultato dell'applicazione della semplificazione. Con C, D vengono indicate clausole generiche, cioè che possono essere sia positive che negative.

Eliminazione delle tautologie:

$$S \cup \{\rightarrow t = t\} \rightarrow S.$$

Sussunzione:

$$S \cup \{C, D\} \rightarrow S \cup \{C\}$$

se C *sussume* D, ovvero se esiste una sostituzione σ tale che $D\sigma \equiv C$.

Demodulazione [WRCS67]:

$$S \cup \{\rightarrow l = r, C\} \rightarrow S \cup \{\rightarrow l = r, C[r\sigma]_p\},$$

dove:

(i) $l\sigma \equiv C|_p$;

(ii) $l\sigma \succ r\sigma$.

Capitolo 4

Implementazione

Questo Capitolo presenta l'implementazione di `auto paramodulation`, la tattica basata su paramodulazione, nell'ambito di HELM e CIC. Come per il resto di HELM, tutto il codice è scritto in linguaggio OCaml[1].

Il Capitolo è diviso in tre parti: nella prima discuteremo degli adattamenti alla teoria della paramodulazione a (un sottoinsieme di) CIC, nonché delle strutture dati usate per la rappresentazione OCaml delle clausole; la seconda parte è invece dedicata alla descrizione dell'algoritmo principale che realizza la tattica; infine, quella conclusiva presenta alcuni risultati, anche con un confronto con la "normale" tattica `auto`.

4.1 La paramodulazione in CIC

CIC è molto più complesso ed espressivo del calcolo dei predicati non tipato, il sistema nel quale la paramodulazione solitamente viene utilizzata. In più, la nostra versione della paramodulazione si applica solamente a teorie puramente equazionali, ai problemi cioè in cui sia il goal che tutti i teoremi e le ipotesi applicabili sono equazioni. È quindi chiaro che la teoria illustrata al Capitolo 3 non può essere applicata così com'è a CIC, ma sono necessari degli adattamenti. In particolare, ciò è stato fatto consentendo l'uso della paramodulazione solo per un sottoinsieme di CIC che corrisponde "più o meno" ad un calcolo del primo

[1] `http://caml.inria.fr`

ordine, cioè a termini in cui possono essere presenti solo astrazioni del primo ordine, ossia di termini su termini che sono argomenti di un'applicazione.

Un po' più formalmente, gli unici termini considerati hanno uno tra i seguenti tipi[2]:

Definizione 4.1 (Termini accettati da `auto paramodulation`).

Teoremi e ipotesi nel contesto locale:

1. prodotto non dipendente, cioè $\Pi x : A.t$ in cui A ha tipo SET e t non dipende da x. In questo caso, useremo la notazione $\rightarrow$ invece di Π: ad esempio, $\Pi dummy : A.A$ verrà indicato con $A \rightarrow A$; oppure
2. applicazione del tipo induttivo `cic:/Coq/Init/Logic/eq.ind`, avente tipo $\Pi A : \text{TYPE}.A \rightarrow A \rightarrow \text{PROP}$, con argomenti un termine A di tipo SET e due termini di tipo A che non contengono astrazioni: termini di questo tipo corrispondono ad equazioni ground. Ad esempio, (`eq` $A\ t_1\ t_2$) corrisponde a $t_1 =_A t_2$, in cui $=_A$ indica l'uguaglianza per termini di tipo A; oppure
3. termine avente la forma

$$\Pi x_1 : A.(\Pi x_2 : A.(\ldots(\Pi x_n : A.(\texttt{eq}\ A\ t_1\ t_2)))),$$

in cui A ha tipo SET e t_1 e t_2 dipendono da $x_1, \ldots, x_n$ ma non contengono astrazioni. Questo tipo di termini corrisponde ad equazioni $t_1 =_A t_2$ in cui t_1 e t_2 contengono le *variabili* $x_1, \ldots, x_n$.

Goal:

applicazione di `cic:/Coq/Init/Logic/eq.ind` ad un termine A di tipo SET e due termini di tipo A che non contengono astrazioni (cioè come al punto 2 qui sopra).

Esempio 4.1. *Ad esempio, alcuni termini ammessi sono:*

1. A : SET
   $x : A$

[2]ricordiamo che HELM si basa sull'isomorfismo di Curry-Howard, quindi goal e teoremi sono visti come tipi abitati dalle rispettive dimostrazioni.

$n : nat$ *(tipo induttivo* `cic:/Coq/Init/Datatypes/nat.ind`*)*
$f_1 : A \to A \to A$ *(funzione a due argomenti di tipo A, ricordiamo che corrisponde al tipo* $\Pi dummy_1 : A.(\Pi dummy_2 : A.A)$*)*
$f_2 : nat \to nat \to nat$
$g : nat \to nat$

2. $(eq\ nat\ (plus\ (S\ 0)\ n)\ (S\ n))$

3. $\lambda x_1 : nat.(\lambda x_2 : nat.(eq\ nat\ (f_2\ x_1\ x_2)\ (g\ n)))$.

Supponiamo ora di volere tradurre in CIC il problema di dimostrare il goal $f(a, b) = g(a)$*, avendo a disposizione l'ipotesi* $f(x_1, x_2) = g(x_1)$*, ed in cui* $\mathcal{F} = \{f, g, a, b\}$ *e* $\mathcal{X} = \{x_1, x_2\}$*. Per prima cosa, dobbiamo assegnare un tipo agli oggetti coinvolti: definiamo quindi un termine A di tipo* SET*, che sarà il tipo delle costanti e delle variabili; quindi procediamo a definire i tipi delle funzioni f e g coerentemente con la loro arietà. In questo modo otteniamo il seguente insieme di termini nel contesto*[3]*:*

$$A : \text{SET}$$
$$a : A$$
$$b : A$$
$$f : A \to A \to A$$
$$g : A \to A$$

A questo punto, dobbiamo tradurre l'ipotesi $f(x_1, x_2) = g(x_1)$*. In accordo al punto 2 della Definizione 4.1, otteniamo*

$$H : \Pi x_1 : A.(\Pi x_2 : A.(eq\ A\ (f\ x_1\ x_2)\ (g\ x_1))),$$

Infine, il goal ha tipo

$$(eq\ A\ (f\ a\ b)\ (g\ a))$$

4.1.1 Variabili e metavariabili

In questo passaggio dal calcolo del primo ordine a CIC, le variabili $\mathcal{X}$ vengono quindi mappate negli identificatori introdotti dall'operatore λ. Per ottenere

[3] ricordiamo che qui ed in seguito con contesto intendiamo il contesto Γ della metavariabile che rappresenta la dimostrazione del goal, si vedano §2.2.4 e §2.3.3

un'istanza ground di un termine che contiene variabili, perciò, è sufficiente applicare il termine agli argomenti opportuni. Quindi, continuando l'Esempio 4.1, applicando H ad a e b, $(H\ a\ b)$, otteniamo un termine di tipo $(eq\ A\ (f\ a\ b)\ (g\ a))$, cioè una dimostrazione per il goal.

Tuttavia, usando la paramodulazione la dimostrazione si ottiene in modo diverso: prima applicando una superposizione sinistra per riscrivere $f(a,b) = g(a)$ in $g(a) = g(a)$, usando $f(x_1, x_2) \Rightarrow g(x_1)$, con $mgu\ \sigma = \{x_1 \mapsto a, x_2 \mapsto b\}$, e quindi applicando la risoluzione di uguaglianza al nuovo goal. Per fare la stessa cosa in CIC, dovremmo unificare i due termini $\Pi x_1 : A.(\Pi x_2 : A.(f\ x_1\ x_2))$ e $(f\ a\ b)$ in modo da poter applicare la riscrittura con la superposizione sinistra. Perché ciò sia possibile, dobbiamo però prima *trasformare le variabili in metavariabili*, in modo da rendere i due termini unificabili:

$$\Pi x_1 : A.(\Pi x_2 : A.(f\ x_1\ x_2) \text{ diventa quindi } (f\ ?1\ ?2).$$

A questo punto, è possibile unificare i due termini ottenendo la sostituzione

$$\sigma = \{?1 \mapsto x_1 : A, ?2 \mapsto x_1 : A\}.$$

Più precisamente, ogni equazione di tipo 3 (rispetto alla Definizione 4.1) viene trasformata in una di tipo 2, in cui gli identificatori $x_1, \ldots, x_n$ vengono sostituiti con delle metavariabili; al nuovo termine così ottenuto viene quindi associato l'appropriato `metasenv` (§2.2.4). Dal punto di vista implementativo, tale operazione è la stessa effettuata dalla tattica `apply`, che applica un teorema ad un goal, ed anche il codice utilizzato è pertanto lo stesso, cioè la funzione `ProofEngineHelpers.saturate_term`.

4.1.2 Rappresentazione delle equazioni

La traduzione in CIC delle equazioni del primo ordine appena descritta è quella che deve effettuare l'utente di Matita per definire un nuovo teorema. Internamente ad `auto paramodulation`, però, le equazioni non sono rappresentate come termini CIC, ma in maniera più strutturata e più adatta alla manipolazione secondo le regole di inferenza e di semplificazione descritte al Capitolo 3.

La struttura dati usata è illustrata in Figura 4.1. Essa è composta dai seguenti campi:

```
type equality =
    int *                (* weight *)
    proof *              (* proof *)
    (Cic.term *          (* type *)
     Cic.term *          (* left side *)
     Cic.term *          (* right side *)
     Utils.comparison) * (* ordering *)
    Cic.metasenv         (* environment for metas *)
```

Figura 4.1: Tipo di dato delle equazioni

weight: Peso dell'equazione, usato per la strategia di selezione (§4.2.2);

proof: Dimostrazione di questa equazione. Non si tratta della prova come termine CIC, ma di un oggetto di tipo `proof`, dal quale la prova come termine CIC verrà effettivamente ricostruita al termine dell'esecuzione di `auto paramodulation`. Sia il tipo `proof` che l'algoritmo per la costruzione delle prove sono oggetto del Capitolo 6;

type: Tipo dell'uguaglianza;

left side: Termine a sinistra dell'=;

right side: Termine a destra dell'=;

ordering: Ordinamento tra `left side` e `right side` (§3.2.2 e §4.2.4);

environment for metas: `metasenv` contenente le metavariabili introdotte come spiegato in §4.1.1;

Rappresentazione delle clausole

In un ambito puramente equazionale, le clausole non sono altro che equazioni con una posizione, a destra o a sinistra di $\rightarrow$ rispettivamente in caso di teoremi (o ipotesi) e goal. Pertanto esse sono rappresentate semplicemente come una coppia $\langle posizione, equazione \rangle$, in cui la posizione è `Negative` per il goal e `Positive` altrimenti.

4.2 L'algoritmo principale

La nozione teorica di ricerca di una dimostrazione per refutazione mediante paramodulazione (e saturazione) che abbiamo discusso fino a questo punto è troppo astratta per essere direttamente implementabile. Per ottenere un algoritmo che la realizzi, dobbiamo prima chiarire tre punti:

1. Come è rappresentato lo spazio di ricerca, cioè l'insieme di clausole;
2. Come sono identificate le possibili inferenze; e
3. Come la prossima inferenza da applicare è scelta tra quelle possibili.

Le risposte a queste domande si concretizzano in un algoritmo noto come *algoritmo given-clause*, che è alla base di tutti i theorem-prover più avanzati, tra i quali OTTER [McC03], SPASS [Wei01], VAMPIRE [Ria03] e WALDMEISTER [HL02]. In particolare, la nostra implementazione si basa sulle descrizioni delle procedure usate da SPASS e VAMPIRE.

La grande popolarità di questo algoritmo è dovuta alla sua semplicità e flessibilità, che consentono facilmente di adattarlo a differenti strategie (§4.2.2 e §4.2.3).

4.2.1 L'algoritmo *given-clause*

L'idea dell'algoritmo è quella di implementare la selezione delle inferenze selezionando una delle clausole coinvolte: selezionando una clausola, tutte le inferenze tra di essa e le altre clausole già selezionate in precedenza vengono abilitate.

Una descrizione ad alto livello della procedura (Figura 4.2) è la seguente: abbiamo due insiemi di clausole, *passive* e *active*. Inizialmente, *active* è vuoto e *passive* contiene tutte le clausole in ingresso, cioè il goal, i teoremi equazionali della libreria[4] e le ipotesi locali. Ad ogni iterazione dell'algoritmo, viene selezionata (tramite la funzione *select*) da *passive* una nuova clausola, chiamata *current*, che viene *attivata*, cioè aggiunta ad *active*. Immediatamente dopo essere stata attivata, *current* viene passata alla funzione *infer*, che ritorna l'insieme (*new*) di tutte le clausole ottenute applicando inferenze tra *current* e tutte le

[4] in realtà, un sottoinsieme di essi costruito come spiegato al Capitolo 7.

```
var new, passive, active: insiemi di clausole
var current: clausola
active := ∅
passive := insieme delle clausole di input
while passive ≠ ∅ do
    current := select(passive)
    passive := passive \ {current}
    active := active ∪ {current}
    new := infer(current, active)
    if new contiene la clausola vuota
        then return "success"
    passive := passive ∪ new
end
return "failure"
```

Figura 4.2: Algoritmo *given-clause* (senza semplificazioni)

clausole in *active*. Se *new* contiene la clausola vuota, la ricerca di una contraddizione termina con successo, altrimenti le nuove clausole ottenute vengono aggiunte a *passive* e si procede con un'altra iterazione.

4.2.2 Strategie di selezione

È noto, e i test effettuati lo confermano ampiamente (§4.3), che uno dei fattori cruciali per le prestazioni è la strategia con cui *current* viene selezionata da *passive*. È purtroppo altresì noto che non esiste una strategia ottimale per ogni tipo di problema, ma al contrario quello che per un particolare goal si rivela essere un metodo molto efficace potrebbe dare pessimi risultati con un altro goal, o con altre ipotesi a disposizione. Per questi motivi all'interno di `auto paramodulation` ne sono state implementate diverse, con la possibilità da parte dell'utente di decidere quale usare e di cambiare i valori di alcuni dei parametri. La strategia usata per default è quella che mediamente si è comportata meglio nei test fin qui effettuati (si veda §4.3).

Prima di descrivere in dettaglio le diverse strategie disponibili, vogliamo ricordarne i requisiti (definiti in §3.2.3): devono essere fair e devono privilegiare

i goal ai teoremi, ossia se *passive* contiene clausole negative, nessuna clausola positiva può essere selezionata.

Selezione cronologica

La strategia più semplice a cui si possa pensare è quella di selezionare sempre la clausola più vecchia, cioè quella che è stata inserita per prima in *passive*. In altre parole, prima si attivano i teoremi e le ipotesi locali, quindi le conseguenze dirette di essi, e così via. L'implementazione di questa strategia è banale: basta infatti rappresentare *passive* come una lista di clausole, in cui gli inserimenti vengono fatti in coda e la selezione consiste nel rimuoverne la testa.

Selezione in base al peso

L'idea di questa strategia è quella di associare ad ogni clausola[5] un *peso*, un intero positivo che ne denota in qualche modo la "complessità". Le clausole vengono quindi mantenute in una coda a priorità, e *select* ritorna sempre quella "più semplice". I termini sono virgolettati perché in effetti la definizione di complessità di una clausola non è univoca, anche se generalmente è legata al numero di sottotermini e (meta)variabili[6] che essa contiene. Questo è anche il nostro caso: la definizione di peso di un'equazione, data formalmente qui sotto, è legata sia alla dimensione (intesa come numero di simboli) dei due lati dell'equazione, sia al numero di variabili che essi contengono. La definizione è presa da Spass.

Definizione 4.2 (Peso di un'equazione). Sia $t : A \equiv l =_A r$ un'equazione. Definiamo *peso di* t, indicato con $w_=(t)$, come

$$w_=(t) = w'(l) + w'(A) + w'(r) + 2 \cdot cv(t),$$

[5] in effetti, ad ogni equazione.

[6] d'ora in avanti, dove non altrimenti specificato considereremo variabile un sinonimo di metavariabile.

dove $w'(u)$ è definita ricorsivamente sui termini CIC come:

$$w'(u) = \begin{cases} 0 & \text{se } u \text{ è una metavariabile} \\ w'(u_1) + \ldots + w'(u_n) & \text{se } u \equiv (u_1 \ldots u_n) \\ w'(t_1) + w'(t_2) + 1 & \text{se } u \equiv \lambda u_1 : t_1.t_2 \text{ o } u \equiv \Pi u_1 : t_1.t_2 \\ 1 & \text{altrimenti} \end{cases}$$

e $cv(t)$ è uguale al numero di variabili distinte in t.

Esempio 4.2. *Ad esempio,* $a =_A a$ *ha peso 3,* $?1 =_A (f\ ?1\ ?2)$ *ha peso 6, e* $(f\ ?1\ (g\ (g\ ?2\ (h\ ?1))\ ?3)) =_A (f\ ?1\ (g\ ?2\ ?3))$ *ha peso 13.*

L'idea alla base di questa strategia di selezione è che le clausole più semplici hanno una maggiore probabilità di essere utilizzate per generarne di nuove tramite paramodulazione e per semplificare quelle più complesse tramite demodulazione e sussunzione (§4.2.3).

Selezione in base alla somiglianza con il goal

L'ultima delle strategie di base implementate è quella che seleziona sempre la clausola "più somigliante" al goal attuale. Essa si applica solo alle clausole positive, pertanto non è usata quando *passive* contiene clausole negative. L'idea è che al crescere della somiglianza tra una clausola ed il goal attuale[7] cresce anche la probabilità che la clausola sia utile alla dimostrazione, sia direttamente, perché può essere usata in un inferenza (o semplificazione) che coinvolge il goal, sia indirettamente, generando altre clausole positive utili.

Il criterio di somiglianza implementato si basa sul numero di sottotermini comuni a due clausole: maggiore è tale numero, maggiore è la somiglianza. Formalmente:

Definizione 4.3 (Somiglianza tra due equazioni). Sia t un termine CIC. Definiamo come *multi insieme dei simboli* di t, $\mathcal{S}_t(t)$, il seguente multi insieme:

$$\mathcal{S}_t(t) = \begin{cases} \emptyset & \text{se } t \text{ è una metavariabile} \\ \mathcal{S}_t(t_1) \cup \ldots \cup \mathcal{S}_t(t_n) & \text{se } t = (t_1 \ldots t_n) \\ \{t\} & \text{altrimenti} \end{cases}$$

[7]Naturalmente, il goal attuale non è sempre quello dato in input, ma potrebbe essere una sua semplificazione ottenuta tramite superposizione o demodulazione.

Il multi insieme dei simboli di un'equazione $e \equiv l = r$ è l'unione dei simboli dei suoi due lati: $\mathcal{S}(e) = \mathcal{S}_t(l) \cup \mathcal{S}_t(r)$.

Siano adesso e_1 ed e_2 due equazioni. Chiamiamo *somiglianza tra* e_1 *ed* e_2, $\|e_1, e_2\|$, il seguente intero:

$$\|e_1, e_2\| = -1 \cdot (|\mathcal{S}(e_1) \triangle \mathcal{S}(e_2)| + \text{abs}(|\mathcal{S}(e_1) \cap \mathcal{S}(e_2)| - |\mathcal{S}(e_1)|)),$$

dove $\triangle$ è la differenza simmetrica tra multi insiemi, $|\cdot|$ la cardinalità di un multi insieme e abs indica il valore assoluto[8].

Esempio 4.3. *Ad esempio, dato il goal* $goal \equiv (f_1\ (f_2\ a\ b)\ (f_3\ a)) = (f_3\ b)$ *e l'equazione* $e \equiv ?1 = (f_2\ c\ (f_3\ ?1))$*, abbiamo*

$$\begin{aligned}
&\mathcal{S}(goal) = \{f_1, f_2, a, b, f_3, a, f_3, b\} \\
&\mathcal{S}(e) = \{f_2, c, f_3\} \\
&\mathcal{S}(goal) \cup \mathcal{S}(e) = \{f_2, f_3\} \\
&\mathcal{S}(goal) \triangle \mathcal{S}(e) = \{f_1, a, b, a, f_3, b\}
\end{aligned}$$

Quindi

$$\|goal, e\| = -1 \cdot (6 + \text{abs}(2 - 8)) = -12.$$

Se invece consideriamo $e_2 \equiv (f_1\ (f_2\ ?1\ ?2)\ (f_3\ ?1)) = (f_3\ ?2)$*, abbiamo*

$$\|goal, e_2\| = -1 \cdot (4 + \text{abs}(4 - 8)) = -8.$$

Data questa definizione, la strategia basata sulla somiglianza consiste nel selezionare la clausola e per cui $\|goal, e\|$[9] è massima. A parità di somiglianza, la preferenza va alle clausole di peso minore.

Questa strategia è stata sviluppata autonomamente, e a quanto ci risulta non è presente in nessuno dei theorem-prover che abbiamo preso in esame.

Combinazioni tra le strategie di base

Le strategie basate su peso e somiglianza non sono utilizzabili direttamente, in quanto violano il requisito di fairness. Per risolvere questo problema, esse

[8] si noti che in generale $\|e_1, e_2\| \neq \|e_2, e_1\|$, ma questo non costituisce un problema in quanto il primo argomento è sempre il goal attuale.

[9] in effetti, $\|g', e'\|$ se $goal \equiv \langle \texttt{Negative}, g' \rangle$ e $e \equiv \langle \texttt{Positive}, e' \rangle$.

vengono combinate con la selezione cronologica, per formare le strategie di *peso-età* e *somiglianza-età*. Queste sono controllate da un intero k che rappresenta il *rapporto* tra le clausole selezionate per peso (risp. somiglianza) e quelle per età: ogni $k + 1$ clausole selezionate, k sono scelte con la strategia basata sul peso (risp. sulla somiglianza), ed una con quella cronologica. In questo modo la fairness è garantita.

In effetti, l'implementazione di *select* in `auto paramodulation` utilizza una singola strategia *somiglianza-peso-età*, costituita dalla combinazione delle tre di base. Essa è parametrica rispetto a due interi k_{wa} e k_{sw}, che possono essere specificati dall'utente, e che indicano rispettivamente il rapporto tra il numero di clausole selezionate per peso o somiglianza e quelle selezionate per età e il rapporto tra quelle selezionate per somiglianza rispetto a quelle selezionate per peso.

Esempio 4.4. *Ad esempio, $k_{wa} = 4$ e $k_{sw} = 3$ significano che ogni cinque clausole selezionate, una verrà scelta in base all'età e le altre quattro con una delle altre strategie, e cioè tre per somiglianza ed una per peso.*

4.2.3 Procedure di semplificazione

La versione "di base" dell'algoritmo *given-clause*, descritta sopra, prevede solamente l'uso delle regole di inferenza, e non di quelle di eliminazione e semplificazione. Tuttavia è non solo possibile, ma anche necessario per ottenere una procedura efficiente estendere l'algoritmo base in modo da includere anche la semplificazione degli insiemi di clausole. Ci sono diverse strategie per fare ciò: nel nostro lavoro di tesi ne abbiamo prese in considerazione due, che chiameremo *full-reduction strategy* e *lazy-reduction strategy* rispettivamente, che sono le più usate dai theorem-prover attuali. In particolare, entrambe sono disponibili sia in SPASS che in VAMPIRE, i sistemi che sono serviti da guida nella realizzazione di `auto paramodulation`.

Full-reduction strategy

In Figura 4.3 è riportato il codice OCaml che implementa la strategia di semplificazione *full-reduction*.

L'idea alla base questa versione dell'algoritmo è quella di mantenere il più ristretto possibile lo spazio di ricerca, *eliminando* tutte le clausole ridondanti o *semplificando* le clausole "complesse" (§3.3.1). La semplificazione avviene utilizzando la regola di *demodulazione*, con cui alcuni sottotermini di un'equazione vengono sostituiti con altri più piccoli rispetto all'ordinamento di riduzione, usando un'altra equazione come regola di riscrittura. L'eliminazione avviene invece utilizzando le regole di *sussunzione* e di *eliminazione di tautologie*. L'intero processo di semplificazione può essere diviso in tre fasi:

1. Quando una clausola è selezionata e diventa *current*, viene sottoposta ad una semplificazione *in avanti*, in cui le equazioni in $active \cup passive$ sono usate per demodulare o eliminare *current*;

2. Subito dopo il passo di inferenza, le nuove clausole in *new* sono soggette a loro volta a semplificazione in avanti, il cui risultato è un nuovo insieme new_s;

3. new_s è quindi usato per operare una semplificazione *all'indietro*, ovvero per demodulare o eliminare le clausole in $active \cup passive$ che sono diventate ridondanti a causa di qualcuna delle clausole in new_s. Durante questa fase, le clausole in $active \cup passive$ che vengono riscritte per demodulazione vengono tolte dal loro insieme di appartenenza e inserite in uno speciale insieme *retained*: se alla fine della semplificazione all'indietro *retained* non è vuoto, il suo contenuto viene aggiunto a new_s, ed il processo riparte dal passo 2 (con $new \equiv new_s \cup retained$), iterando la semplificazione finché non si ottiene $retained = \emptyset$, che indica che tutta la ridondanza[10] è stata eliminata.

In questa versione di *given-clause*, le clausole in *passive* sono passive solo nel senso che non sono utilizzate nelle regole di inferenza, ma esse giocano comunque un ruolo significativo nella procedura di semplificazione.

[10] secondo le regole definite in §3.3.1: ci possono infatti essere clausole ridondanti che non vengono identificate come tali dalle regole di semplificazione utilizzate.

Lazy-reduction strategy

Abbiamo detto che l'idea principale alla base del *given-clause* con *full-reduction* è quella di mantenere l'insieme delle clausole il più ridotto possibile, eliminando tutta la ridondanza. Questa operazione è tuttavia decisamente costosa dal punto di vista computazionale: tipicamente infatti l'insieme *passive* cresce molto velocemente, e questo fa sì che da un certo punto in avanti dell'esecuzione (e abbastanza presto in effetti) il tempo speso per semplificare l'insieme *passive* domini quello speso per le inferenze e le altre semplificazioni.

Questa osservazione è alla base della versione che abbiamo chiamato *lazy-reduction* di *given-clause*, la cui implementazione è visibile in Figura 4.4. Essa si differenzia dalla precedente perché le clausole in *passive* sono effettivamente "passive", cioè non vengono utilizzate né per le inferenze né per la semplificazione di *active* e *new*. Ciò consente, a parità di tempo, di generare molte più clausole rispetto all'algoritmo di *full-reduction*, tuttavia il prezzo da pagare è una ridondanza potenzialmente molto maggiore, che causa un allargamento dello spazio di ricerca. Come abbiamo già sottolineato altrove, purtroppo non c'è in generale una strategia che sia migliore dell'altra: per alcuni goal potrebbe funzionare meglio la strategia *full*, mentre per altri quella *lazy*. Pertanto, `auto paramodulation` le implementa entrambe, consentendo all'utente di decidere quale usare.

4.2.4 Ordinamento dei termini

Un altro fattore che influenza in maniera sensibile le prestazioni della tattica[11] è l'ordinamento di riduzione tra i termini. Dei quattro presentati al Capitolo 3 (§3.2.2), ne abbiamo implementati tre: LPO e le due varianti di KBO (standard e non ricorsiva).

Tutti e tre si basano sulla stessa precedenza $\succ_{\mathcal{F}}$:

Definizione 4.4 (Precedenza $\succ_{\mathcal{F}}$). La *precedenza* $\succ_{\mathcal{F}}$ è una relazione d'ordine sui termini ground (ovvero senza metavariabili) definita come segue:

- se t_1 e t_2 sono identificatori (sia locali che riferiti al contesto), con i_1 e i_2

[11] una stima del grado di influenza dei singoli fattori è oggetto della prossima Sezione, §4.3.

i loro indici di DeBrujin, allora

$$t_1 \succ_{\mathcal{F}} t_2 \iff i_1 < i_2;$$

altrimenti

- se t_2 è un identificatore e t_1 no, allora $t_1 \succ_{\mathcal{F}} t_2$; altrimenti
- se t_1 e t_2 sono costanti, il loro ordinamento è quello indotto dall'ordine lessicografico dei loro uri; altrimenti
- se t_2 è una costante e t_1 no, $t_1 \succ_{\mathcal{F}} t_2$; altrimenti
- se t_1 e t_2 sono tipi induttivi, il loro ordinamento è quello indotto dall'ordine lessicografico dei loro uri; altrimenti
- se t_2 è un tipo induttivo e t_1 no, $t_1 \succ_{\mathcal{F}} t_2$; altrimenti
- se t_1 e t_2 sono costruttori di tipi induttivi, il loro ordinamento è quello indotto dall'ordine lessicografico dei loro uri; altrimenti
- se t_2 è un costruttore di tipo induttivo e t_1 no, $t_1 \succ_{\mathcal{F}} t_2$; altrimenti
- se $t_1 \equiv (u_1, \ldots, u_n)$ e $t_2 \equiv (v_1, \ldots, v_n)$, allora

$$t_1 \succ_{\mathcal{F}} t_2 \iff u_1 \succ_{\mathcal{F}} v_1$$

- in tutti gli altri casi, t_1 e t_2 non sono confrontabili.[12]

[12] a rigore, $\succ_{\mathcal{F}}$ dovrebbe essere un ordinamento totale. Tuttavia, in pratica ciò non dovrebbe costituire un problema, visto che (i) i casi precedenti coprono la grandissima parte dei casi pratici (tutti quelli incontrati), e (ii) l'ordine è comunque ben fondato.

```
let rec given_clause_fullred env passive active =
  match passive_is_empty passive with
  | true -> ParamodulationFailure
  | false ->
      let (sign, current), passive = select env passive active in
      let res = forward_simplify env (sign, current) ~passive active in
      match res with
      | None -> given_clause_fullred env passive active
      | Some (sign, current) ->
          if (sign = Negative) && (is_identity env current) then
            ParamodulationSuccess (Some current, env)
          else
            let new' = infer env sign current active in
            let active =
              if is_identity env current then active
              else
                let al, tbl = active in
                match sign with
                | Negative -> (sign, current)::al, tbl
                | Positive ->
                   al @ [(sign, current)], Indexing.index tbl current
            in
            let rec simplify new' active passive =
              let new' = forward_simplify_new env new' ~passive active in
              let active, passive, newa, retained =
                backward_simplify env new' ~passive active in
              match newa, retained with
              | None, None -> active, passive, new'
              | Some (n, p), None
              | None, Some (n, p) ->
                  let nn, np = new' in
                  simplify (nn @ n, np @ p) active passive
              | Some (n, p), Some (rn, rp) ->
                  let nn, np = new' in
                  simplify (nn @ n @ rn, np @ p @ rp) active passive
            in
            let active, passive, new' = simplify new' active passive in
            match contains_empty env new' with
            | false, _ ->
                let passive = add_to_passive passive new' in
                given_clause_fullred env passive active
            | true, goal -> ParamodulationSuccess (goal, env)
```

Figura 4.3: Algoritmo *given-clause* con *full-reduction*

```
let rec given_clause_lazyred env passive active =
  match passive_is_empty passive with
  | true -> ParamodulationFailure
  | false ->
      let (sign, current), passive = select env passive active in
      let res = forward_simplify env (sign, current) ~passive active in
      match res with
      | None -> given_clause_lazyred env passive active
      | Some (sign, current) ->
          if (sign = Negative) && (is_identity env current) then
            ParamodulationSuccess (Some current, env)
          else
            let new' = infer env sign current active in
            let res, goal = contains_empty env new' in
            if res then ParamodulationSuccess (goal, env) else
              let new' = forward_simplify_new env new' active in
              let active =
                match sign with
                | Negative -> active
                | Positive -> let active, _, newa, _ =
                      backward_simplify env ([], [current]) active in
                    match newa with
                    | None -> active
                    | Some (n, p) ->
                        let al, tbl = active in
                        let nn = List.map (fun e -> Negative, e) n in
                        let pp, tbl =
                          List.fold_right
                            (fun e (l, t) -> (Positive, e)::l,
                                             Indexing.index tbl e)
                            p ([], tbl)
                        in (nn @ al @ pp, tbl)  in
              match contains_empty env new' with
              | false, _ ->
                  let active =
                    let al, tbl = active in
                    match sign with
                    | Negative -> (sign, current)::al, tbl
                    | Positive ->
                        al @ [(sign, current)], Indexing.index tbl current in
                  let passive = add_to_passive passive new' in
                  given_clause_lazyred env passive active
              | true, goal -> ParamodulationSuccess (goal, env)
```

Figura 4.4: Algoritmo *given-clause* con *lazy-reduction*

4.3 Risultati ottenuti

Questa Sezione è dedicata alla presentazione di alcuni risultati sperimentali relativi alle prestazioni di `auto paramodulation`. Il contenuto è suddiviso in tre parti: nella prima mostreremo come le diverse opzioni configurabili (strategia di selezione e di semplificazione e ordinamento) influenzino anche sensibilmente l'efficienza della tattica, quindi nelle altre due tratteremo il confronto della nuova tattica rispettivamente con l'`auto` "tradizionale" e con SPASS, scelto come rappresentante dei theorem-prover più avanzati. Prima di procedere, facciamo notare che i risultati qui riportati risentono anche delle ottimizzazioni al codice che verranno discusse nel prossimo Capitolo. Tutti i tempi indicati si riferiscono a test effettuati su di un Pentium III 700Mhz con 256Mb di RAM, con sistema operativo GNU/Linux Slackware 10.1, ed utilizzando il compilatore nativo `ocamlopt` per compilare il codice OCaml.

4.3.1 Differenze tra le diverse configurazioni

In questa Sezione vogliamo mostrare con degli esempi concreti quale sia l'impatto dell'ordinamento di riduzione e delle strategie di selezione delle clausole e di riduzione della ridondanza sull'efficienza della tattica. Gli esempi qui proposti sono significativi in quanto dimostrano chiaramente che non esistono scelte ottimali per questi tre parametri, ma al contrario la scelta migliore dipende dal goal in esame.

Criteri di ordinamento

1. Il primo esempio che presentiamo è tratto dalla libreria TPTP [SS01], una collezione di problemi per la valutazione e il confronto delle prestazioni di theorem-prover per la logica del primo ordine. Si tratta del problema `BOOO03-2`, che chiede di dimostrare l'idempotenza della moltiplicazione in una logica booleana, cioè che per ogni x vale

   $$x * x = x.$$

   Utilizzando un ordinamento LPO, la dimostrazione viene prodotta in 2.12 secondi (con *full-reduction* e selezione delle clausole data da $k_{wa} = 3$ e

$k_{sw} = 2$), mentre utilizzando KBO il tempo scende a 0.68 e 0.54 secondi rispettivamente per la versione standard e non-ricorsiva.

2. Su di un problema molto simile, `BOO004-2` (idempotenza dell'addizione), con una strategia *lazy-reduction*, abbiamo che la scelta di LPO consente di ottenere la soluzione in 2.6 secondi, con KBO standard il tempo si dimezza circa (1.35 secondi), mentre con KBO non ricorsivo si raggiunge il tempo limite di 60 secondi senza aver concluso la ricerca. Semplicemente cambiando la strategia di semplificazione a *full-reduction*, i risultati cambiano drasticamente: 10.92 secondi per LPO, 1.14 per KBO standard e 1.21 per quello non ricorsivo. Ciò evidenzia un'ulteriore complicazione, e cioè che i tre parametri non sono nemmeno indipendenti tra loro.

Strategia di selezione

In tutti i test seguenti, l'ordinamento usato è KBO non ricorsivo, e la strategia di semplificazione è *full-reduction*.

1. Per il problema `BOO001-1`, un problema di logica booleana che chiede di dimostrare che

$$(x^{-1})^{-1} = x,$$

i valori $k_{wa} = 4$ e $k_{sw} = 0$ portano ad una soluzione in 1.41 secondi; basta tuttavia portare k_{wa} a 5 per ottenere un tempo molto maggiore: 3.21 secondi. Con $k_{wa} = 3$ e $k_{sw} = 2$ otteniamo invece un tempo di 2.15 secondi.

2. La situazione si capovolge per il problema seguente: dimostrare che in una logica booleana (le ipotesi sono le stesse di `BOO003-2`), per ogni x e y, vale

$$x^{-1} = (x + y)^{-1} + x^{-1}.$$

Con $k_{wa} = 2$ e $k_{sw} = 1$, la dimostrazione viene trovata in 1.53 secondi, con $k_{wa} = 3$ e $k_{sw} = 2$ il tempo sale a circa 17 secondi, e con $k_{wa} = 4$ e $k_{sw} = 0$ a circa 43.

Strategia di semplificazione

Concludiamo con un'analisi dell'impatto delle diverse strategie di semplificazione. Gli esempi sono stati testati con ordinamento KBO non ricorsivo e strategia di selezione data da $k_{wa} = 3$ e $k_{sw} = 2$.

1. L'esempio 3 precedente, che viene risolto in 17 secondi circa con *full-reduction*, viene completato con *lazy-reduction* in 1.03 secondi.

2. Al contrario, il problema `BOOOO4-2` come abbiamo già visto in precedenza non termina entro 60 secondi con *lazy-reduction*, mentre con *full-reduction* viene completato in 1.21 secondi.

La mutua dipendenza dei parametri dal goal in esame, dei parametri tra di loro e delle prestazioni dai parametri è certamente un problema non trascurabile. Tuttavia, esso non è dovuto all'implementazione di `auto paramodulation`, ma è condiviso da tutti i theorem-prover basati sulla saturazione (e in §4.3.3 ne mostreremo un esempio). Nonostante ciò, i test effettuati hanno consentito di determinare dei valori di default che forniscono mediamente delle prestazioni accettabili in tutti i casi presi in esame finora: selezione delle clausole con $k_{wa} = 3$ e $k_{sw} = 2$, ordinamento KBO non ricorsivo e strategia di semplificazione *full-reduction*.

4.3.2 Confronto con `auto`

Se escludiamo il tempo richiesto per determinare i teoremi della libreria applicabili al goal (si veda il Capitolo 7), che comunque è lo stesso per entrambe le tattiche, `auto paramodulation` è sensibilmente più veloce di `auto`, almeno di un ordine di grandezza nei casi più sfavorevoli, ed in media gli ordini di grandezza di differenza sono due o più. Il tempo di accesso alla libreria è comunque di qualche secondo: pertanto esso diventa significativo solo per problemi facili per entrambe le tattiche, per i quali le differenze tra le due sarebbero comunque poco significative. Per problemi più complessi, invece, solitamente il tempo richiesto da `auto` è dominante rispetto all'accesso alla libreria, per cui le considerazioni fatte sopra restano valide.

Come esempi, riportiamo tre problemi significativi perché mostrano grandi differenze di prestazioni quando sono dati in input ad `auto`. In tutti e tre i casi, `auto paramodulation` usa i settaggi di default.

1. Il primo problema chiede di dimostrare che per ogni n naturale, si ha:

   $$(S\ n) * (S\ n) = (S\ (n + n + n * n)),$$

   dove S è la funzione successore. Il contesto non contiene nessuna ipotesi, ma si usano solo i teoremi di libreria.

   Tempo di `auto paramodulation`: 0.076 secondi.

   Tempo di `auto`: 122.12 secondi (con $width = 5$ e $depth = 3$, cioè i valori predefiniti[13]).

2. Il secondo problema è ancora una volta un problema sui naturali, la cui dimostrazione utilizza solamente i teoremi di libreria. Il goal da dimostrare è che per ogni n naturale:

   $$n + n = 2 * n.$$

   Tempo di `auto paramodulation`: 0.052 secondi.

   Tempo di `auto`: 208.58 secondi con la configurazione predefinita, 25.46 secondi con $width = 3$ e $depth = 3$.

3. Infine, l'ultimo esempio è un problema tratto dalla libreria TPTP: il problema esaminato è quello denominato `COL004-3`. Date una funzione di applicazione *app* e due termini S e K, sotto le ipotesi

   $$\begin{array}{ll} (app\ (app\ (app\ S\ x)\ y)\ z) = (app\ (app\ x\ z)\ (app\ y\ z)) & \forall x, y, z, \text{ e} \\ (app\ (app\ K\ x)\ y) = x & \forall x, y \end{array}$$

   si vuole dimostrare che per ogni x e y

   $$(app\ (app\ (app\ (app\ S\ (app\ K\ (app\ S\ I)))\ (app\ (app\ S\ I)\ I))\ x)\ y) = (app\ y\ (app\ (app\ x\ x)\ y))),$$

[13] *width* e *depth* sono due parametri che controllano la dimensione dello spazio di ricerca per `auto`. Valori bassi indicano che l'albero dei sottogoal aperti sarà potato spesso, quindi consentono una terminazione più veloce, ma se i valori sono troppo bassi la potatura potrebbe essere eccessiva e non permettere quindi di trovare una dimostrazione.

dove I è un'abbreviazione per $(app\ (app\ S\ K)\ K)$. In questo caso, `auto paramodulation` non necessita della libreria per dimostrare il goal (`auto` sì perché ha bisogno degli assiomi che definiscono $=$).

Tempo di `auto paramodulation`: 0.077 secondi.

Tempo di `auto`: *fallimento* con la configurazione standard. Nessuna combinazione di valori di *width* e *depth* ha comunque portato al completamento della dimostrazione entro 5 minuti.

Queste enormi differenze di prestazioni sono da attribuirsi quasi esclusivamente all'uso della saturazione in luogo di una definizione esplicita dell'$=$ con gli assiomi di Figura 3.1. In particolare, il problema è dato specialmente dalla transitività, la cui applicazione incontrollata causa la generazione di un elevatissimo numero di sottogoal. Naturalmente, occorre notare che `auto` ha un campo di applicazione molto più ampio di `auto paramodulation`, non essendo limitata a teorie puramente equazionali, tuttavia riteniamo che le differenze di prestazioni compensino questa limitazione della presente tattica[14].

Infine, osserviamo che le procedure di semplificazione giocano un ruolo cruciale nelle prestazioni di `auto paramodulation`:

Esempio 4.5. *Per il primo problema qui sopra la tattica trova nella un insieme di teoremi applicabili che contiene alcuni teoremi ridondanti. Ad esempio, il teorema*

$$\forall n, m, p \quad (n * m) * p = n * (m * p)$$

è ridondante se sono già presenti i teoremi

$$\forall n, m, p \quad n * (m * p) = (n * m) * p \quad e$$
$$\forall n, m \quad n * m = m * n.$$

Se questa ridondanza non viene eliminata tramite una pre-semplificazione dei teoremi applicabili (operazione normalmente effettuata da `auto paramodulation`*), il tempo impiegato per trovare una dimostrazione cresce notevolmente: circa 17 secondi con $k_{wa} = 5$ e $k_{sw} = 0$.*

[14] la limitazione, inoltre, non è necessariamente destinata a permanere, come discuteremo nel Capitolo 8.

4.3.3 Confronto con Spass

Per verificare le prestazioni della nostra implementazione rispetto ai più avanzati theorem-prover per la logica del primo ordine, abbiamo scelto di confrontare `auto paramodulation` con SPASS. Il motivo di questa scelta è duplice: innanzitutto SPASS è uno dei theorem-prover più efficienti, ed inoltre sia il suo codice sorgente che un'ampia base di test (una copia della libreria TPTP[SS01] già tradotta nella sintassi di SPASS) si possono facilmente reperire dalla home page del progetto[15].

Come previsto, i risultati sono a favore di SPASS, tuttavia le differenze nei test finora effettuati sono mediamente di un solo ordine di grandezza, con casi a noi favorevoli in cui il divario si riduce ad un fattore 3-4.

Negli esempi che seguono, dove non altrimenti specificato sia SPASS che `auto paramodulation` sono stati testati con i settaggi di default.

1. Come primo esempio, consideriamo nuovamente il problema `COL004-3` già presentato nel confronto con `auto`.

   Tempo di `auto paramodulation`: 0.077 secondi.

   Tempo di SPASS: 0.01 secondi.

   Clausole generate da `auto paramodulation`: 15 in totale, 10 non ridondanti.

   Clausole generate da SPASS: 3 non ridondanti.

2. Il secondo esempio che riportiamo è `LAT029-1`, che dice che il *meet* di a e a è sempre a. Gli assiomi a disposizione sono:

   $$\forall X, Y : (join\ (meet\ X\ Y)\ (meet\ X\ (join\ X\ Y))) = X$$
   $$\forall X, Y : (join\ (meet\ X\ X)\ (meet\ Y\ (join\ X\ X))) = X$$
   $$\forall X, Y : (join\ (meet\ X\ Y)\ (meet\ Y\ (join\ X\ Y))) = Y$$
   $$\forall X, Y, Z : (meet\ (meet\ (join\ X\ Y)\ (join\ Z\ X))\ X) = X$$
   $$\forall X, Y, Z : (join\ (join\ (meet\ X\ Y)\ (meet\ Z\ X))\ X) = X$$

   ed il goal è

   $$(meet\ a\ a) = a.$$

[15] `http://spass.mpi-sb.mpg.de/`

Tempo di `auto paramodulation`: 3.99 secondi.

Tempo di SPASS: 0.14 secondi.

Clausole generate da `auto paramodulation`: 237 totali, 87 non ridondanti.

Clausole generate da SPASS: 285 di cui 149 non ridondanti.

3. Il terzo esempio è `BOO001-1` ($(x^{-1})^{-1} = x$).

   Tempo di `auto paramodulation`: 2.11 secondi con *full-reduction*, 0.83 con *lazy-reduction*.

   Tempo di SPASS: 0.12 secondi.

   Clausole generate da `auto paramodulation`: 238 totali, 96 non ridondanti con *full-reduction*; 281 di cui 134 non ridondanti con *lazy-reduction*.

   Clausole generate da SPASS: 272 di cui 86 non ridondanti.

4. Come quarto esempio, vogliamo considerare `ROB002-1`, un problema nell'algebra di Robbins che dice che se $-(-x) = x$ allora l'algebra è booleana; in altri termini, il goal da dimostrare è

   $$-(x + (-y)) + (-((-x) + (-y))) = y$$

   supponendo che

   $$-(-x) = x$$

   ed avendo a disposizione gli assiomi di commutatività e distributività dell'addizione e l'assioma di Robbins

   $$\forall x, y : \ -(-(x + y) + (-(x + (-y)))) = x.$$

   Tempo di `auto paramodulation`: 0.99 secondi.

   Tempo di SPASS: 0.12 secondi.

   Clausole generate da `auto paramodulation`: 103 totali, 40 non ridondanti.

   Clausole generate da SPASS: 256 di cui 59 non ridondanti.

5. L'ultimo esempio che presentiamo è `BOOO15-4`, significativo perché può essere considerato un problema difficile per entrambi i programmi, come mostrano i tempi di esecuzione. Il goal in questo caso è dimostrare in una logica booleana la legge di DeMorgan

$$(x * y)^{-1} = x^{-1} + y^{-1}.$$

Tempo di `auto paramodulation`: 539 secondi.

Tempo di SPASS: 36.29 secondi. Tuttavia, cambiando il valore di *WDRatio* (che regola la strategia di selezione, si veda [Wei01] per i dettagli) da 5 (default) a 3, il tempo sale a circa 72 secondi: ciò per dimostrare come anche l'efficienza di SPASS sia molto sensibile alle strategie di selezione e semplificazione scelte.

Clausole generate da `auto paramodulation`: 4919 totali, 953 non ridondanti.

Clausole generate da SPASS: 22388 di cui 3266 non ridondanti.

Nonostante la differenza nei tempi di esecuzione, questi esempi evidenziano come le dimensioni degli spazi di ricerca (gli insiemi di clausole) siano pressoché identiche, ed anzi in qualche caso (come il problema 5) `auto paramodulation` si comporti notevolmente meglio nel contenere il numero di clausole generate nella ricerca della soluzione. Ciò ci fa supporre che la maggiore velocità di SPASS sia dovuta in gran parte ad una più efficiente implementazione, e non all'utilizzo di tecniche più sofisticate. Considerando che:

- la nostra implementazione utilizza il più possibile codice già presente in HELM, che non è stato progettato con il theorem-proving in mente, e pertanto molte delle sue strutture dati e funzioni non sono ottimizzate per questo scopo, ma piuttosto per una facile manipolabilità, manutenibilità, estensibilità e modularità;

- OCaml, nonostante sia molto veloce quando compilato in codice nativo, è comunque tipicamente più lento del C (linguaggio in cui SPASS è scritto);

- SPASS è uno dei theorem-prover più avanzati, con anni di sviluppo e ottimizzazioni alle spalle, mentre il nostro codice ha solo qualche mese di vita;

- nonostante i termini CIC trattati dalla nostra tattica siano solo un sottoinsieme dei possibili (§4.1), CIC stesso è comunque molto più complesso di un calcolo del primo ordine non tipato, pertanto gli algoritmi che operano sui termini sono intrinsecamente più complicati nel nostro caso;

riteniamo che le prestazioni attuali siano molto incoraggianti.

Capitolo 5

Dettagli implementativi e ottimizzazioni

L'obbiettivo di questo Capitolo è di illustrare i principali problemi riscontrati nell'implementazione della tattica, e le soluzioni adottate per risolverli. Un po' grossolanamente, esse si possono dividere in tre categorie: ottimizzazioni "semplici", ovvero che riducono i tempi di un fattore costante, ottimizzazioni "sostanziali", tali cioè da rendere trattabili problemi che altrimenti non portrebbero essere affrontati (per problemi non solo di tempo ma anche di occupazione di memoria), e soluzioni a problemi di correttezza nati dall'adattamento della paramodulazione a CIC.

5.1 Trattamento delle metavariabili

Le regole di inferenza e di semplificazione definite al Capitolo 3 assumono implicitamente che le clausole coinvolte non abbiano nessuna variabile in comune. Normalmente i theorem-prover garantiscono ciò semplicemente impedendo la condivisione di variabili tra clausole: ad esempio, in $\rightarrow f(x, y, z) = g(z)$ e $\rightarrow g(x) = h(x, y)$, x e y nella prima clausola non hanno nulla a che fare con le variabili con lo stesso nome nella seconda.

Nel passaggio a CIC, però, nel quale le variabili vengono tradotte con delle metavariabili, la condizione di non condivisione deve essere garantita esplicitamente, mappando le variabili in clausole diverse in metavariabili diver-

se: il codice di HELM, infatti, ed in particolare per quanto concerne `auto paramodulation` l'algoritmo di unificazione, considerano sempre due metavariabili con lo stesso indice come la stessa metavariabile. Questo significa che ogni volta che si applicano le regole di superposizione, la nuova clausola ottenuta deve essere post-processata per garantire che non contenga metavariabili già in uso.

Esempio 5.1. *Ad esempio, applicando una superposizione destra a*

$$\rightarrow (f\ ?1\ (g\ ?1\ ?2)) = (h\ ?2)$$

usando

$$\rightarrow (g\ ?3\ ?4) \Rightarrow ?4,$$

otterremmo

$$\rightarrow (f\ ?1\ ?2) = (h\ ?2),$$

avendo $\sigma = \{?3 \mapsto ?1, ?4 \mapsto ?2\}$. *Poiché però le clausole non possono condividere metavariabili, le metavariabili nella nuova clausola devono essere sostituite, ottenendo quindi*

$$\rightarrow (f\ ?5\ ?6) = (h\ ?6).$$

Questa operazione di sostituzione delle metavariabili già in uso con metavariabili nuove ha luogo dopo ogni inferenza per superposizione, ed è compiuta dalla funzione `Inference.fix_metas`.

5.2 Unificazione e matching

Le regole di semplificazione (sussunzione e demodulazione, §3.3.1) hanno tra le loro condizioni vincoli del tipo

$$t_2\sigma \equiv t_1,$$

in cui cioè la sostituzione σ istanzia le variabili di t_2 in modo da renderlo uguale a t_1. Se σ esiste, si dice che t_1 è un'*istanza* di t_2. Il problema di determinare σ è detto *matching*: si tratta di una versione ristretta dell'unificazione, in cui l'unificatore può istanziare solamente le variabili del primo termine.

Pur essendo un'operazione assolutamente analoga all'unificazione, tuttavia, all'interno di HELM il matching non era implementato. Essendo l'algoritmo

di unificazione per termini CIC abbastanza complicato [Sac04, Dow01], anziché modificare il codice esistente per adattarlo al caso particolare del matching, la soluzione adottata consiste nel cercare di unificare t_1 e t_2, e in caso di successo verificare quindi che la sostituzione σ ritornata sia compatibile con l'operazione di matching, ovvero o istanzia solo le metavariabili di t_2, oppure istanzia metavariabili di t_1 solo con altre metavariabili. In quest'ultimo caso, la sostituzione viene modificata sostituendo tutti i binding $?1 \mapsto ?2$ in cui $?1 \in t_1$ con $?2 \mapsto ?1$. Il codice della funzione `matching` è riportato in Figura 5.1.

Esempio 5.2. *Ad esempio,* $(f\ ?1\ (g\ a\ b))$ *è un'istanza di* $(f\ ?4\ ?5)$*. Per come è implementata, l'unificazione di HELM ritorna sempre sostituzioni in cui la metavariabile con indice minore è mappata in quella con indice maggiore, in questo caso si avrebbe quindi*

$$\sigma = \{?1 \mapsto ?4, ?5 \mapsto (g\ a\ b)\}.$$

σ deve essere quindi "corretta" in

$$\sigma' = \{?4 \mapsto ?1, ?5 \mapsto (g\ a\ b)\}.$$

Un'implementazione di questo tipo, che prima tenta un'unificazione normale e poi sistema a posteriori il risultato, è sicuramente meno efficiente di una che invece sfrutti una versione specializzata dell'algoritmo di unificazione. Tuttavia questa scelta ha anche un vantaggio, oltre alla maggiore semplicità e chiarezza: ottimizzando l'unificazione generale, anche il matching ne risente positivamente. Ciò è esattamente quanto è stato fatto nel nostro caso.

5.2.1 Unificazione semplificata

L'unificazione in CIC è molto più complessa di quella del primo ordine [BS01, Dow01]. In particolare, quando una metavariabile viene unificata con un termine, devono essere unificati anche i loro tipi, ed inoltre quando vengono unificate due metavariabili esse devono essere ristrette allo stesso contesto (§2.2.4). Nel calcolo dei predicati (non tipato) queste due operazioni non sono necessarie: per unificare una variabile x con un termine t è sufficiente verificare che t non contenga x, e quindi restituire l'unificatore $\sigma = \{x \mapsto t\}$. Analogamente, se

```
let matching metasenv context t1 t2 ugraph =
    try
      let subst, metasenv, ugraph =
        unification metasenv context t1 t2 ugraph
      in
      let t' = CicMetaSubst.apply_subst subst t1 in
      if not (meta_convertibility t1 t') then
        raise MatchingFailure
      else
        let metas = metas_of_term t1 in
        let fix_subst = function
          | (i, (c, Cic.Meta (j, lc), ty)) when List.mem i metas ->
              (j, (c, Cic.Meta (i, lc), ty))
          | s -> s
        in
        let subst = List.map fix_subst subst in
        subst, metasenv, ugraph
    with
    | CicUnification.UnificationFailure _
    | CicUnification.Uncertain _ ->
      raise MatchingFailure
;;
```

Figura 5.1: Codice dell'algoritmo di matching

sapessimo già che i termini da unificare hanno lo stesso tipo, che questo tipo non contiene metavariabili, e che le metavariabili nei due termini hanno tutte lo stesso contesto, le operazioni di unificazione dei tipi e restrizione dei contesti potrebbero essere evitate. In effetti, dalla Definizione 4.1 sappiamo che i tipi dei termini trattati da `auto paramodulation` non contengono metavariabili e che tutte le metavariabili presenti nelle equazioni hanno lo stesso contesto. Inoltre, un controllo preliminare garantisce che l'unificazione venga chiamata solo per termini dello stesso tipo. Ciò rende possibile utilizzare all'interno di `auto paramodulation` un algoritmo di unificazione semplificato, che si comporta esattamente come quello del primo ordine [BS01], e che ha tempi di esecuzione sensibilmente inferiori a `CicUnification.fo_unif`, l'algoritmo di unificazione per il caso più generale di due termini CIC qualunque. Il codice della funzione

`unification_simple` è riportato in Figura 5.2.

Esempio 5.3. *Ad esempio, consideriamo il problema* `BOOOO4-2` *della libreria TPTP (idempotenza dell'addizione in una logica booleana). Usando la normale unificazione (*`CicUnification.fo_unif`*), il tempo impiegato a completare la dimostrazione è 3.26 secondi; usando* `unification_simple`*, invece, il tempo scende a 1.23 secondi. La differenza di prestazioni è ancora più evidente per il goal* $x^{-1} = (x + y)^{-1} + x^{-1}$ *(sempre in una logica booleana), che viene risolto in 17 secondi usando* `unification_simple` *e in circa 53 con* `CicUnification.fo_unif`*.*

5.3 Indicizzazione

Uno dei maggiori problemi prestazionali del theorem-proving basato su paramodulazione è data dal rapido incremento delle dimensioni degli insiemi di clausole da trattare. Nonostante le procedure di semplificazione ed eliminazione delle ridondanze riducano notevolmente lo spazio di ricerca, esso tuttavia continua a crescere nel tempo, e per problemi non banali il numero di clausole negli insiemi *active* e *passive* raggiunge l'ordine delle decine di migliaia in pochi minuti di esecuzione. Appare quindi chiaro che le operazioni che comprendono la ricerca di clausole o termini, come la ricerca di termini unificabili o istanze di un termine dato per applicare le regole di inferenza o di semplificazione, non possono essere implementate con algoritmi di ricerca lineare sugli insiemi di termini, ma che è necessaria una qualche forma di *indicizzazione* di tali insiemi che permetta di rendere il tempo di ricerca idealmente indipendente dalla dimensione degli insiemi, o comunque meno sensibile all'incremento di essa. In effetti, tutti i theorem-prover attuali implementano diverse forme di indicizzazione, spesso specializzate per compiere una singola operazione il più velocemente possibile [SRV01]. In questa tesi, abbiamo implementato due delle tecniche di indicizzazione più comuni, *path indexing* e *discrimination tree* (usati in varie forme ad esempio anche da Otter, Vampire e Waldmeister) analizzandone quindi le prestazioni per decidere quale delle due utilizzare in `auto paramodulation`. Entrambe le tecniche si basano sulle stesse idee, che verranno introdotte nella prossima sezione preliminare.

```
let unification_simple metasenv context t1 t2 ugraph =
  let rec unif subst menv s t =
    let s = match s with C.Meta _ -> lookup s subst | _ -> s
    and t = match t with C.Meta _ -> lookup t subst | _ -> t in
    match s, t with
    | s, t when s = t -> subst, menv
    | C.Meta (i, _), C.Meta (j, _) when i > j -> unif subst menv t s
    | C.Meta _, t when occurs_check subst s t ->
        raise (U.UnificationFailure "Inference.unification.unif")
    | C.Meta (i, l), t -> (
        try
          let _, _, ty = CicUtil.lookup_meta i menv in
          let subst =
            if not (List.mem_assoc i subst) then
               (i, (context, t, ty))::subst
            else subst
          in subst, menv
        with CicUtil.Meta_not_found m -> assert false
      )
    | _, C.Meta _ -> unif subst menv t s
    | C.Appl (hds::_), C.Appl (hdt::_) when hds <> hdt ->
        raise (U.UnificationFailure "Inference.unification.unif")
    | C.Appl (hds::tls), C.Appl (hdt::tlt) -> (
        try
          List.fold_left2
            (fun (subst', menv) s t -> unif subst' menv s t)
            (subst, menv) tls tlt
        with Invalid_argument _ ->
          raise (U.UnificationFailure "Inference.unification.unif")
      )
    | _, _ -> raise (U.UnificationFailure "Inference.unification.unif")
  in
  let subst, menv = unif [] metasenv t1 t2 in
  let menv =
    List.filter
      (fun (m, _, _) ->
         try let _ = List.find (fun (i, _) -> m = i) subst in false
         with Not_found -> true) menv
  in List.rev subst, menv, ugraph
```

Figura 5.2: Codice dell'algoritmo di unificazione semplificato

5.3.1 Caratteristiche comuni e idee generali

L'idea alla base delle tecniche di indicizzazione qui trattate è la seguente: per ogni clausola[1] $\rightarrow l = r$ si costruiscono le *rappresentazioni come stringhe* dei due termini l ed r; su queste stringhe si possono quindi applicare noti algoritmi di *string-matching* per ottenere un insieme di tutti i termini che sono in relazione (unificazione, matching, uguaglianza strutturale) con il *termine di query*. In particolare, le stringhe che rappresentano i termini (*position strings*, o *p-string*) vengono inserite in un *trie*, in cui ciascuna foglia contiene un insieme di clausole che contengono termine corrispondente alla stringa che identifica il cammino dalla radice alla foglia del trie (si veda [SRV01] per i dettagli). *Path indexing* e *discrimination tree* si differenziano per come sono costruite le *p-string* che rappresentano i termini e come queste sono inserite nel trie.

Definizione 5.1 (*p-string*). Sia t un termine CIC. Una *position string* (o *p-string*) per t è una sequenza non vuota $\langle p_1, s_1\rangle\langle p_2, s_2\rangle \ldots \langle p_n, s_n\rangle$, in cui i p_i sono posizioni (v. Definizione 3.3) e gli s_i sono termini o il simbolo speciale '$*$', definita nel seguente modo:

- per ogni $1 \leq i, j \leq n$, se p_i è un prefisso proprio di p_j, allora $i < j$;
- per ogni $1 \leq i \leq n$ abbiamo che $root(t|_{p_i}) = s_i$, dove $root(u)$ è definita come:

$$root(u) = \begin{cases} * & \text{se } u \text{ è una metavariabile} \\ u_1 & \text{se } u \equiv (u_1, \ldots, u_m) \\ u & \text{altrimenti} \end{cases}$$

Esempio 5.4. *Ad esempio, se*

$$t \equiv (f\ ?1\ (g\ ?2\ c)),$$

alcune p-string *per* t *sono:*

$$\langle \Lambda, f\rangle\langle 1, *\rangle\langle 2, g\rangle\langle 2.1, *\rangle\langle 2.2, c\rangle,$$
$$\langle \Lambda, f\rangle\langle 1, *\rangle, e$$
$$\langle \Lambda, f\rangle\langle 2, g\rangle\langle 2.2, c\rangle$$

[1] positiva. Le clausole negative (i goal) non vengono indicizzate.

In pratica, le posizioni $p_1, \ldots, p_n$ in una p-string *per un termine t rappresentano un modo di attraversare t, ed in questo caso $s_1, \ldots, s_n$ sono i sottotermini visitati in questo attraversamento.*

Operazioni sugli indici

Le operazioni compiute sugli indici sono le seguenti:

Ricerca: dato un termine di query t, questa operazione ritorna l'insieme di clausole che soddisfano una certa relazione R con t. In `auto paramodulation`, R può essere:

- *unificazione:* $R(\rightarrow l = r, t)$ vale se $l\sigma = t\sigma \vee r\sigma = t\sigma$ per qualche sostituzione σ;
- *matching:* $R(\rightarrow l = r, t)$ vale se $l\sigma = t \vee r\sigma = t$ per qualche sostituzione σ.

In effetti, sia *path indexing* che la versione dei *discrimination tree* qui descritta supportano solo ricerche approssimate, ovvero se $S = \{c | R(c, t)\}$ è l'insieme di clausole in relazione con t, le ricerche ritornano un insieme $S' \supseteq S$, che deve poi essere filtrato (con l'idea che le operazioni per estrarre S' da tutto l'insieme di clausole siano molto più veloci di quelle richieste per estrarre S da S') per ottenere S.

Inserimento di una clausola: per inserire $\rightarrow l = r$ nell'indice, innanzitutto si creano le rappresentazioni come stringhe di l ed r. Queste vengono quindi cercate nel trie (oppure inserite se non già presenti), e la clausola viene quindi aggiunta all'insieme associato al nodo ritornato dalla ricerca (o inserimento). Se l ed r sono ordinati rispetto all'ordinamento di riduzione, cioè $l \succ r$ oppure $r \succ l$, si inserisce nel trie solo il termine maggiore.

Rimozione di una clausola: per rimuovere $\rightarrow l = r$ dall'indice, si rimuove la clausola dagli insiemi contenuti nei nodi ritornati dalla ricerca di (*p-strings* rappresentanti) l ed r nel trie. Se uno di questi insiemi diventa vuoto, si rimuove anche la *p-string* corrispondente.

Costruzione dell'indice: essa si limita a creare un nuovo trie vuoto.

5.3.2 Path indexing

Per questo tipo di indicizzazione, i termini sono associati ad un insieme di *p-string*, ciascuna rappresentante un attraversamento del termine t dalla radice ad una foglia, dove per radice intendiamo $root(t)$ (v. Definizione 5.1) e per foglia un sottotermine u per cui $root(u) = u$ oppure una metavariabile. Considerando solo *p-string* di questo tipo, la Definizione 5.1 può essere semplificata: infatti, anziché specificare per ogni sottotermine la sua posizione relativamente al termine t di partenza, è sufficiente indicarne la posizione relativa rispetto al termine genitore.

Esempio 5.5. *Consideriamo il termine* $t \equiv (f\ b\ (g\ (f\ ?1)\ a))$. *La* p-string *da* $f\ a\ ?1$ *sarebbe:*

$$\langle \Lambda, f\rangle\langle 2, g\rangle\langle 2.1, f\rangle\langle 2.1.1, *\rangle.$$

Con la rappresentazione semplificata, essa diventa:

$$f.2.g.1.f.1.*\,.$$

L'insieme di p-string *che identifica* t *è:*

$$\{f.1.b, \quad f.2.g.1.f.1.*, \quad f.2.g.2.a\}.$$

Esempio di indice

Consideriamo il seguente insieme di clausole[2]:

$$\rightarrow (f\ (g\ a\ ?1)\ c) = ?1 \quad (1) \qquad \rightarrow (f\ (g\ ?2\ b)\ ?3) = (g\ ?3\ ?2) \quad (2)$$
$$\rightarrow (f\ (g\ a\ b)\ c) = b \quad (3) \qquad \rightarrow (f\ (g\ ?4\ c)\ b) = (f\ ?4\ c) \quad (4)$$

In Figura 5.3 è mostrato il *path index* per l'insieme.

Operazioni di ricerca

Le Figure 5.4 e 5.5 riportano il codice delle funzioni di ricerca per unificazione e matching rispettivamente.

Esaminando il codice possiamo osservare che la ricerca non necessita di *backtracking*: ogni sottotermine del termine di query è esaminato una sola volta, e le

[2]le clausole sono tali che il termine a sinistra dell'= sia maggiore di quello a destra. Inoltre, per questo esempio è irrilevante che l'insieme contenga clausole ridondanti.

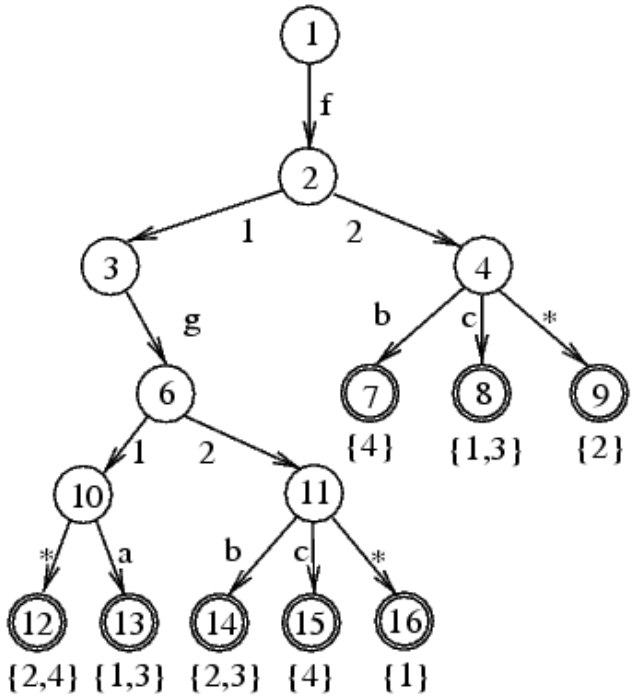

Figura 5.3: Esempio di *path index*

prestazioni ne risentono positivamente. In effetti, il collo di bottiglia di entrambi gli algoritmi è un altro, e precisamente la combinazione dei risultati intermedi, in particolare l'intersezione tra insiemi di clausole.

5.3.3 Discrimination tree

Nei *discrimination tree* ogni termine è rappresentato da una sola *p-string*, ottenuta con un attraversamento del termine in ordine prefisso. Come per il *path indexing*, anche in questo caso possiamo sfruttare la natura delle *p-string* considerate per darne una definizione semplificata ed ottimizzata per i *discrimination tree*. Ciò può essere fatto osservando che, dato che le applicazioni $(t_1 \dots t_n)$ hanno una arietà fissa, la stringa ottenuta con una scansione in preordine del termine è univocamente determinata anche ignorando le posizioni. Pertanto esse possono essere rimosse senza problemi.

Esempio 5.6. *Ad esempio, la* p-string *per il termine* $t \equiv (f\ b\ (g\ (f\ ?1\ a)))$

```
let rec retrieve_unifiables trie term =
  match trie with
  | PSTrie.Node (value, map) ->
      let res =
        match term with
        | Cic.Meta _ ->
            PSTrie.fold
              (fun ps v res -> PosEqSet.union res v)
              (PSTrie.Node (None, map)) PosEqSet.empty
        | _ ->
            let hd_term = head_of_term term in
            try
              let n = PSMap.find (Term hd_term) map in
              match n with
              | PSTrie.Node (Some v, _) -> v
              | PSTrie.Node (None, m) ->
                  let l =
                    PSMap.fold
                      (fun k v res ->
                         match k with
                         | Index i ->
                             let t = subterm_at_pos i term in
                             let s = retrieve_unifiables v t in s::res
                         | _ -> res) m [] in
                  match l with
                  | hd::tl ->
                      List.fold_left (fun r s -> PosEqSet.inter r s)
                        hd tl
                  | _ -> PosEqSet.empty
            with Not_found ->
              PosEqSet.empty in
      try
        let n = PSMap.find (Term (Cic.Implicit None)) map in
        match n with
        | PSTrie.Node (Some s, _) -> PosEqSet.union res s
        | _ -> res
      with Not_found -> res
```

Figura 5.4: Codice della ricerca per unificazione in un *path index*

```
let rec retrieve_generalizations trie term =
  match trie with
  | PSTrie.Node (value, map) ->
      let res =
        match term with
        | Cic.Meta _ -> PosEqSet.empty
        | term ->
            let hd_term = head_of_term term in
            try
              let n = PSMap.find (Term hd_term) map in
              match n with
              | PSTrie.Node (Some s, _) -> s
              | PSTrie.Node (None, m) ->
                  let l =
                    PSMap.fold
                      (fun k v res ->
                         match k with
                         | Index i ->
                             let t = subterm_at_pos i term in
                             let s = retrieve_generalizations v t in
                             s::res
                         | _ -> res) m []
                  in
                  match l with
                  | hd::tl ->
                      List.fold_left (fun r s -> PosEqSet.inter r s)
                        hd tl
                  | _ -> PosEqSet.empty
            with Not_found ->
              PosEqSet.empty in
      try
        let n = PSMap.find (Term (Cic.Implicit None)) map in
        match n with
        | PSTrie.Node (Some s, _) -> PosEqSet.union res s
        | _ -> res
      with Not_found -> res
```

Figura 5.5: Codice della ricerca per matching in un *path index*

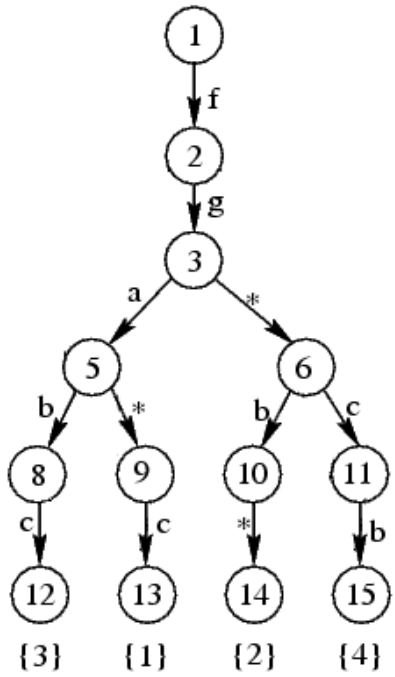

Figura 5.6: Esempio di *discrimination tree*

sarebbe:

$$\langle \Lambda, f \rangle \langle 1, b \rangle \langle 2, g \rangle \langle 2.1, f \rangle \langle 2.1.1, * \rangle \langle 2.1.2, a \rangle.$$

Rimuovendo gli indici, essa si semplifica in:

$$f.b.g.f.*.a.$$

Esempio di indice

In Figura 5.6 è mostrato il *discrimination tree* per l'insieme di clausole della Sezione precedente, ovvero:

$$\rightarrow (f\ (g\ a\ ?1)\ c) = ?1 \quad (1) \qquad \rightarrow (f\ (g\ ?2\ b)\ ?3) = (g\ ?3\ ?2) \quad (2)$$
$$\rightarrow (f\ (g\ a\ b)\ c) = b \quad (3) \qquad \rightarrow (f\ (g\ ?4\ c)\ b) = (f\ ?4\ c) \quad (4)$$

Operazioni di ricerca

Le Figure 5.7 e 5.8 riportano il codice delle funzioni di ricerca per unificazione e matching nel caso dei *discrimination tree*.

Diversamente dal *path indexing*, in questo caso entrambe le operazioni non richiedono le potenzialmente costose intersezioni tra insiemi di clausole per combinare i risultati intermedi. Tuttavia in questo caso ad ogni passo la posizione successiva da visitare è implicita e deve essere calcolata partendo dalla posizione corrente e dal termine di query. Ciò viene fatto dalle funzioni `next_t` e `after_t`, la cui semantica è la seguente: vedendo il termine t come un albero, se $t|_p = u$, $t|_{next\_t(p,t)}$ è il sottotermine di t visitato immediatamente dopo u, e $t|_{after\_t(p,t)}$ è il sottotermine visitato immediatamente dopo aver attraversato tutti i sottotermini di u.

Esempio 5.7. *Ad esempio, se*

$$t \equiv (f\ (g\ a\ b)\ c),$$

allora

$$t|_{next\_t(1,t)} \equiv a,\ \ e$$
$$t|_{after\_t(1,t)} \equiv c.$$

(dove $t|_1 = g$*).*

Infine, sempre a causa dell'ordine di attraversamento implicito, la ricerca per unificazione fa uso di un'altra funzione ausiliaria, `jump_list`, necessaria quando il sottotermine del termine di query alla posizione corrente è una metavariabile. In questo caso, se alla posizione corrispondente ci sono termini nell'indice che contengono un sottotermine diverso da una metavariabile, dobbiamo "saltare" questo sottotermine: `jump_list(T)`, dove `T` è un sottoalbero del *discrimination tree*, ritorna quindi la lista dei sottoalberi radicati nei nodi corrispondenti alle posizioni immediatamente dopo tutte le posizioni interne al sottotermine corrente.

```
let retrieve_unifiables tree term =
  let rec retrieve tree term pos =
    match tree with
    | DiscriminationTree.Node (Some s, _) when pos = [] -> s
    | DiscriminationTree.Node (_, map) ->
        let subterm =
          try Some (subterm_at_pos pos term) with Not_found -> None in
        match subterm with
        | None -> PosEqSet.empty
        | Some (Cic.Meta _) ->
            let newpos = try next_t pos term with Not_found -> [] in
            let jl = jump_list tree in
            List.fold_left
              (fun r s -> PosEqSet.union r s)
              PosEqSet.empty
              (List.map (fun t -> retrieve t term newpos) jl)
        | Some subterm ->
            let res =
              try
                let hd_term = head_of_term subterm in
                let n = PSMap.find hd_term map in
                match n with
                | DiscriminationTree.Node (Some s, _) -> s
                | DiscriminationTree.Node (None, _) ->
                    retrieve n term (next_t pos term)
              with Not_found -> PosEqSet.empty in
            try
              let n = PSMap.find (Cic.Implicit None) map in
              let newpos = try after_t pos term with Not_found -> [-1]
              in if newpos = [-1] then
                match n with
                | DiscriminationTree.Node (Some s, _) ->
                   PosEqSet.union s res
                | _ -> res
              else
                PosEqSet.union res (retrieve n term newpos)
            with Not_found -> res
  in retrieve tree term []
```

Figura 5.7: Codice della ricerca per unificazione in un *discrimination tree*

```
let retrieve_generalizations tree term =
  let rec retrieve tree term pos =
    match tree with
    | DiscriminationTree.Node (Some s, _) when pos = [] -> s
    | DiscriminationTree.Node (_, map) ->
        let res =
          try
            let hd_term = head_of_term (subterm_at_pos pos term) in
            let n = PSMap.find hd_term map in
            match n with
            | DiscriminationTree.Node (Some s, _) -> s
            | DiscriminationTree.Node (None, _) ->
                let newpos = try next_t pos term with Not_found -> []
                in retrieve n term newpos
          with Not_found ->
            PosEqSet.empty
        in
        try
          let n = PSMap.find (Cic.Implicit None) map in
          let newpos = try after_t pos term with Not_found -> [-1] in
          if newpos = [-1] then
            match n with
            | DiscriminationTree.Node (Some s, _) ->
               PosEqSet.union s res
            | _ -> res
          else
            PosEqSet.union res (retrieve n term newpos)
        with Not_found ->
          res
  in
  retrieve tree term []
```

Figura 5.8: Codice della ricerca per matching in un *discrimination tree*

Problema	Tempo di esecuzione (in secondi)		
	Senza indice	Path indexing	Discrimination tree
TPTP `BOO003-2`	0.89	0.66	0.59
TPTP `BOO004-2`	6.04	1.4	1.25
TPTP `BOO001-1`	8.68	2.84	2.06
$x^{-1} = (x + y)^{-1} + x^{-1}$	56.87	18.11	17.23
TPTP `BOO015-4`	non testato	810	539

Figura 5.9: Prestazioni delle diverse tecniche di indicizzazione

5.3.4 Confronto tra le due tecniche

In Figura 5.9 sono riassunti i risultati dei test effettuati per confrontare *path indexing*, *discrimination tree* ed un approccio senza indicizzazione. Si può vedere chiaramente come l'utilizzo di un indice abbia un impatto positivo sulle prestazioni già per problemi di piccola taglia. Per quanto riguarda il confronto tra i due tipi di indice, per problemi di dimensioni contenute essi sono sostanzialmente equivalenti, tuttavia possiamo notare che quando gli insiemi di clausole raggiungono dimensioni nell'ordine delle migliaia i *discrimination tree* risultano essere più efficaci; per questo essi sono stati scelti per l'implementazione di `auto paramodulation`.

Capitolo 6

La costruzione delle prove

Nei theorem prover basati su risoluzione e paramodulazione, come Otter, Spass o Vampire, la dimostrazione del goal è costituita solitamente da una lista delle clausole generate che termina con la clausola vuota (quella cioè che ha permesso la refutazione del goal negato). Per ogni clausola è tipicamente indicata la giustificazione, ovvero se essa è un'ipotesi iniziale o se è stata ottenuta per inferenza o per semplificazione, ed in questi casi quali sono le clausole e la regola di inferenza (semplificazione) che l'hanno generata. Un esempio di questo tipo di dimostrazione, generato da Otter per il problema `BOOOO1-1` della libreria TPTP, è riportato in Figura 6.1.

Una soluzione analoga non poteva tuttavia essere adottata nel nostro contesto: come detto al Capitolo 2, infatti, Matita si basa sull'isomorfismo di Curry-Howard, quindi una dimostrazione per un goal deve essere un termine CIC che abiti il tipo costituito dal goal stesso. Questo capitolo è dedicato quindi ad illustrare come tale termine venga costruito da `auto paramodulation`.

6.1 Dimostrazione di un passo di riscrittura

La dimostrazione del goal è costruita in maniera incrementale, partendo dalle ipotesi e dai teoremi della libreria e dimostrando ogni passo di inferenza o di semplificazione. I "componenti di base" con cui sono costruite le prove sono le dimostrazioni dei singoli passi di riscrittura: le dimostrazioni, cioè, di $T(a) = u$

```
1 [] finverse(finverse(ca))!=ca.
3 [] fmultiply(fmultiply(x,y,z),u,fmultiply(x,y,v))=
                            fmultiply(x,y,fmultiply(z,u,v)).
5 [] fmultiply(x,y,y)=y.
7 [] fmultiply(x,x,y)=x.
11 [] fmultiply(x,y,finverse(y))=x.
13 [para_into,3.1.1.1,11.1.1] fmultiply(x,y,fmultiply(x,z,u))=
                            fmultiply(x,z,fmultiply(finverse(z),y,u)).
16 [para_into,3.1.1.1,5.1.1] fmultiply(x,y,fmultiply(z,x,u))=
                                    fmultiply(z,x,fmultiply(x,y,u)).
39 [para_into,16.1.1.3,11.1.1] fmultiply(x,y,z)=
                           fmultiply(z,x,fmultiply(x,y,finverse(x))).
57,56 [para_into,39.1.1,16.1.1,flip.1]
        fmultiply(fmultiply(x,y,z),y,fmultiply(y,u,finverse(y)))=
               fmultiply(x,y,fmultiply(y,u,z)).
80 [para_into,13.1.1,7.1.1,flip.1]
        fmultiply(x,y,fmultiply(finverse(y),x,z))=x.
126 [para_into,80.1.1,39.1.1,demod,57]
         fmultiply(finverse(x),y,fmultiply(y,x,z))=y.
242 [para_into,126.1.1.3,5.1.1] fmultiply(finverse(x),y,x)=y.
264 [para_into,242.1.1,11.1.1] finverse(finverse(x))=x.
266 [binary,264.1,1.1] $F.
```

Figura 6.1: Dimostrazione trovata da OTTER per il problema `BOO001-1`

avendo $T(b) = u$ ed $a = b$, in cui $T(a)$ indica un termine T che contiene a come sottotermine.

Esse si ottengono applicando il teorema `cic:/Coq/Init/Logic/eq_ind.con`, che ha il tipo CIC

$$\forall A : \text{TYPE}.\forall x : A.\forall P : (A \to \text{PROP}).(P\ x) \to \forall y : A.x = y \to (P\ y)$$

che dice appunto che posso ottenere una dimostrazione che il predicato P vale per y dalle dimostrazioni che P vale per x e che $x = y$.

6.2 Dimostrazione di un goal

Quando `auto paramodulation` viene invocata, la dimostrazione del goal è una metavariabile, che rappresenta appunto la prova non ancora completata (§2.2.4). La dimostrazione viene costruita combinando le prove dei singoli passi di riscrittura nel modo seguente: supponiamo di avere un'equazione $a = c$ (di tipo A), la cui dimostrazione è P_1 ed un goal $T_1(a) = T_1(c)$, la cui dimostrazione è una metavariabile $?n$. Se viene applicata una superposizione sinistra o una demodulazione, possiamo ottenere $T_1(c) = T_1(c)$, che è una tautologia la cui dimostrazione si ottiene per riflessività (applicando cioè `cic:/Coq/Init/Logic/eq.ind#xpointer(1/1/1)`, ossia `refl_equal`):

$$(\texttt{refl\_equal}\ T_1(c)).$$

Osservando che la tautologia $T_1(c) = T_1(c)$ è equivalente[1] a $((\lambda x : A.T_1(x) = T_1(c))\quad c)$, che quindi ha la stessa dimostrazione, possiamo dimostrare l'equazione di partenza $T_1(a) = T_1(c)$ istanziando la metavariabile $?n$ con un'applicazione di `eq_ind` nel modo seguente:

$$?n \mapsto (\texttt{eq\_ind}\ A\ c\ \lambda x : A.T_1(x) = T_1(c)\ (\texttt{refl\_equal}\ T_1(c))\ a\ P_1),$$

Generalizzado questo procedimento, ogni volta che riscriviamo un goal $G_1(l)$ con dimostrazione $?n_1$ usando un'equazione $l \Rightarrow r$ (di tipo A) con dimostrazione P, creiamo una nuova metavariabile $?n_2$ che rappresenta la dimostrazione del nuovo goal $G_1(r)$, e istanziamo $?n_1$ con

$$(\texttt{eq\_ind}\ A\ r\ \lambda x : A.G_1(x)\ ?n_2\ l\ P).$$

Quando arriviamo a riscrivere il goal in una tautologia $t = t$, l'ultima metavariabile creata, $?n_m$, verrà istanziata con $(\texttt{refl\_equal}\ t)$.

6.2.1 Dimostrazione delle clausole positive

L'ultimo punto da considerare per la costruzione della dimostrazione è come ottenere le prove delle clausole positive generate. Il procedimento è simile a quello

[1] è convertibile, §2.2.2

per i goal: le prove sono composizioni di passi di riscrittura e di applicazioni di teoremi e ipotesi nel contesto.

Il caso base è la dimostrazione diretta per applicazione di un teorema o di un'ipotesi: il termine di prova si ottiene in questo caso semplicemente applicando l'ipotesi/teorema agli argomenti argomenti opportuni. Un semplice esempio chiarisce meglio quanto appena esposto:

Esempio 6.1. *Se ho un ipotesi* $H : \Pi x : A.\Pi y : A.(f\ x\ y)$*, il termine che dimostra* $(f\ a\ b)$ *(con* $f : A \to A \to \textsc{Prop}$*,* $a : A$ *e* $b : A$*), è semplicemente* $(H\ a\ b\ )$*.*

Avendo a disposizione i termini di prova per i teoremi e le ipotesi nel contesto, è possibile costruire induttivamente le dimostrazioni per le equazioni generate applicando demodulazione o superposizione destra, applicando direttamente `eq_ind`: se $l \Rightarrow r$ è l'equazione usata per riscrivere $T(l)$ in $T(r)$, il termine CIC che dimostra $T(r)$ è

$$(\texttt{eq\_ind}\ A\ l\ \lambda x : A.T(x)\ P_{T(l)}\ r\ P_{l=r}),$$

dove A è il tipo dell'uguaglianza, e $P_{T(l)}$ e $P_{l=r}$ sono le dimostrazioni rispettivamente di $T(l)$ e di $l = r$.

6.3 Aspetti implementativi

Dal punto di vista implementativo, ogni oggetto `equality` (Capitolo 4, Figura 4.1) contiene la dimostrazione dell'equazione che rappresenta. Inizialmente essa era già il termine CIC di prova, tuttavia successivamente l'implementazione è cambiata per problemi di efficienza e di utilizzo di memoria. La dimensione delle prove è infatti proporzionale al numero di passi di inferenza/semplificazione applicati per generare l'equazione: questo fa sì che man mano che l'esecuzione procede si generino prove di dimensioni sempre maggiori, il che porta sia il consumo di memoria, sia il tempo impiegato a costruire il termine di prova, a crescere molto rapidamente, anche in considerazione del fatto che il numero di clausole in *active* e *passive* aumenta con il tempo. Inoltre, la grande maggioranza delle clausole generate non viene poi utilizzata per la dimostrazione del goal: le loro prove possono quindi essere considerate del tutto inutili.

```
type proof =
  | BasicProof of Cic.term
  | ProofBlock of
      Cic.substitution * UriManager.uri *
        (* name,    ty,        eq_ty,     left,      right *)
        (Cic.name * Cic.term * Cic.term * Cic.term * Cic.term) *
        (Utils.pos * equality) * proof
  | ProofGoalBlock of proof * equality
  | ProofSymBlock of Cic.term Cic.explicit_named_substitution * proof
```

Figura 6.2: Definizione del tipo di dato `proof`

La soluzione adottata è basata sulla seguente idea: anziché costruire immediatamente il termine di prova ogni volta che si genera una nuova equazione, è sufficiente ricordare come la nuova prova deve essere costruita partendo da quelle delle equazioni da cui quella corrente è stata ottenuta; al termine dell'esecuzione, quindi, le informazioni raccolte possono essere utilizzate per generare solamente la prova per il goal di partenza.

Concretamente, abbiamo definito un tipo di dato `proof` (Figura 6.2) i cui oggetti indicano in che modo le dimostrazioni devono essere costruite. Un oggetto `proof` ha uno dei seguenti significati:

BasicProof: l'oggetto è già una prova completa espressa come termine CIC. I teoremi e le ipotesi locali hanno prove di questo tipo;

ProofBlock: contiene le informazioni per costruire la dimostrazione di una riscrittura. In `ProofBlock(subst, eq_URI, t, eqp, eqproof)`,

- `subst` è la sostituzione ritornata dalla regola di inferenza/semplificazione usata per ottenere l'equazione,
- `eq_URI` è il teorema da applicare per costruire la prova per riscrittura: esso può essere o `cic:/Coq/Init/Logic/eq_ind.con` oppure anche `cic:/Coq/Init/Logic/eq_ind_r.con`, una variante di `eq_ind` che si differenzia da esso solo per l'ordine degli argomenti nell'uguaglianza richiesta per l'applicazione della riscrittura (esso richiede una prova di $y = x$ anziché di $x = y$, si veda §6.1),

`t` è una tupla `(name, ty, eq_ty, left, right)` usata per costruire il termine $\lambda x : A.T(x)$ richiesto per applicare `eq_ind` (si veda la Sezione precedente, §6.2.1),

`eqp` è una tupla `(pos, eq)` in cui `eq` è l'equazione $l = r$ usata come regola di riscrittura e `pos` indica l'orientamento della riscrittura, ovvero se $l \Rightarrow r$ oppure $r \Rightarrow l$,

`eqproof` è la `proof` dell'equazione alla quale è stata applicata la riscrittura che ha generato quella corrente.

ProofGoalBlock: contiene le informazioni per costruire la dimostrazione di un'equazione negativa. In `ProofGoalBlock(proofbit, equality)`,

`proofbit` è la `proof` per la nuova equazione (una `BasicProof` con l'applicazione di `refl_equal`, un `ProofBlock` oppure un altro `ProofGoalBlock`),

`equality` è il goal da cui questo deriva.

ProofSymBlock: permette di ottenere una dimostrazione di $l = r$ a partire da quella di $r = l$, applicando `cic:/Coq/Init/Logic/sym_eq.con`.

6.3.1 Algoritmo di costruzione della prova

L'algoritmo di costruzione della prova è implementato dalla funzione `Inference.build_proof_term`, il cui codice è riportato in Figura 6.3. Il suo funzionamento è illustrato dal seguente esempio.

Esempio 6.2 (Costruzione di una prova). *Supponiamo di voler dimostrare $z = b$ avendo come ipotesi*

$$a \Rightarrow z \qquad a \Rightarrow c \qquad c \Rightarrow b,$$

in cui a, b, c, z hanno tipo A. Ciò può essere fatto applicando una serie di riscritture, $a = z \to a = b \to c = b \to b = b$ fino a giungere alla tautologia $b = b$ che ci permette di concludere. La `proof` per $b = b$ ha la seguente forma[2]*:*

[2]come si può notare, quanto riportato non è esattamente quanto descritto sopra: per semplicità di esposizione, infatti, tutti gli argomenti non necessari di `ProofBlock` sono stati omessi, e il secondo argomento di `ProofGoalBlock` non è un `equality` ma direttamente la `proof` corrispondente.

```
ProofGoalBlock(
    BasicProof (refl_equal b),
    ProofGoalBlock(
        BasicProof ?4,
        ProofGoalBlock(
            ProofBlock(BasicProof ?3),
            ProofGoalBlock(
                ProofBlock(BasicProof ?2),
                ProofBlock(BasicProof ?1)))))
```

Gli oggetti `BasicProof ?n` *sono le nuove metavariabili introdotte con il procedimento illustrato alla Sezione 6.2; esaminando la Figura 6.3, vediamo che il termine che dimostra* $z = b$ *viene costruito nel modo seguente. Innanzitutto si "scende" lungo la* ***proof*** *per rimpiazzare ogni metavariabile con il suo corpo, che viene costruito durante questa "discesa":* `BasicProof ?4` *viene sostituita da* `BasicProof (refl_equal b)`, `ProofBlock(BasicProof ?3)` *da* `ProofBlock(BasicProof (refl_equal b))`, *e così via fino ad ottenere*

```
ProofBlock(ProofBlock(ProofBlock(BasicProof (refl_equal b)))).
```

A questo punto, quindi, si costruisce il termine come composizione delle singole riscritture, invocando ricorsivamente la funzione (e quindi in qualche modo "risalendo" la ***proof****) sui vari* `ProofBlock` *interni.*

In Figura 6.4 è riportato il termine costruito con l'algoritmo appena descritto per il problema `BOO001-1` della libreria TPTP. Nonostante la dimostrazione trovata sia piuttosto lunga, essa tuttavia non è troppo difficile da comprendere: ad esempio, possiamo vedere che il goal (cioè `(inverse (inverse a)) = a`) si dimostra riscrivendo

```
(eq A a (multiply (inverse (inverse a)) a (inverse a)))        (6.1)
```

in

```
(eq A a (inverse (inverse a)))                                 (6.2)
```

```
let build_proof_term equality =
  let rec do_build_proof proof =
    match proof with
    | BasicProof term -> term
    | ProofGoalBlock (proofbit, equality) ->
        let _, proof, _, _, _ = equality in
        do_build_goal_proof proofbit proof
    | ProofSymBlock (ens, proof) ->
        let proof = do_build_proof proof in
        Cic.Appl [Cic.Const (HelmLibraryObjects.Logic.sym_eq_URI, ens);
                  proof]
    | ProofBlock (subst, eq_URI, t', (pos, eq), eqproof) ->
        let name, ty, eq_ty, left, right = t' in
        let bo =
          Cic.Appl [Cic.MutInd (HelmLibraryObjects.Logic.eq_URI, 0, []);
                    eq_ty; left; right] in
        let t' = Cic.Lambda (name, ty, bo) in
        let proof' =
          let _, proof', _, _, _ = eq in do_build_proof proof' in
        let eqproof = do_build_proof eqproof in
        let _, _, (ty, what, other, _), menv', args' = eq in
        let what, other =
          if pos = Utils.Left then what, other else other, what in
        CicMetaSubst.apply_subst subst
          (Cic.Appl [Cic.Const (eq_URI, []); ty;
                     what; t'; eqproof; other; proof'])
  and do_build_goal_proof proofbit proof =
    match proof with
    | ProofGoalBlock (pb, eq) ->
        do_build_proof (ProofGoalBlock (replace_proof proofbit pb, eq))
    | _ -> do_build_proof (replace_proof proofbit proof)
  and replace_proof newproof = function
    | ProofBlock (subst, eq_URI, t', poseq, eqproof) ->
        let eqproof' = replace_proof newproof eqproof in
        ProofBlock (subst, eq_URI, t', poseq, eqproof')
    | ProofGoalBlock (pb, equality) ->
        let pb' = replace_proof newproof pb in
        ProofGoalBlock (pb', equality)
    | BasicProof _ -> newproof
    | p -> p
  in let _, proof, _, _, _ = equality in do_build_proof proof
```

Figura 6.3: Codice della funzione `Inference.build_proof_term`

usando come regola di riscrittura

```
(eq A (multiply (inverse (inverse a)) a (inverse a))
                                        (inverse (inverse a))).
```
(6.3)

(6.1) a sua volta si ottiene a partire da

```
(eq A a (multiply (inverse (inverse a)) a
    (multiply a (inverse a) (inverse a))))
```
(6.4)

usando come regola di riscrittura

```
(eq A (multiply a (inverse a) (inverse a)) (inverse a)),
```
(6.5)

e così via ...

```
(eq_ind_r A
  a
  [x_Demod_241:A](eq A x_Demod_241 a)
  (refl_equal A a)
  (inverse (inverse a))
  (sym_eq{A:=A ; x:=a ; y:=(inverse (inverse a))}
    (eq_ind A
      (multiply (inverse (inverse a)) a (inverse a))
      [x_SupR_256:A](eq A a x_SupR_256)
      (eq_ind A
        (multiply a (inverse a) (inverse a))
        [x_SupR_164:A](eq A a (multiply (inverse (inverse a)) a x_SupR_164))
        (eq_ind_r A
          (multiply a (inverse a)
             (multiply (inverse (inverse a)) a (inverse a)))
          [x_SupR_106:A](eq A x_SupR_106 (multiply (inverse (inverse a)) a
                         (multiply a (inverse a) (inverse a))))
          (eq_ind A
            (multiply (inverse (inverse a)) a a)
            [x_SupR_12:A](eq A (multiply x_SupR_12 (inverse a)
                           (multiply (inverse (inverse a)) a (inverse a)))
                           (multiply (inverse (inverse a)) a
                           (multiply a (inverse a) (inverse a))))
            (H (inverse (inverse a)) a a (inverse a) (inverse a))
            a
            (H1 a (inverse (inverse a))))
          a
          (eq_ind A
            (multiply a (inverse a) (inverse (inverse a)))
            [x_Demod_23:A](eq A x_Demod_23 (multiply a (inverse a)
                           (multiply (inverse (inverse a)) a (inverse a))))
            (eq_ind A
              (multiply a (inverse a) (inverse (inverse a)))
              [x_SupR_42:A](eq A (multiply a (inverse a)
                             (inverse (inverse a))) (multiply a (inverse a)
                             (multiply (inverse (inverse a)) x_SupR_42 (inverse a))))
              (eq_ind A
                (multiply (multiply a (inverse a) (inverse (inverse a)))
                  (multiply a (inverse a) (inverse (inverse a)))
                  (multiply a (inverse a) (inverse a)))
                [x_SupR_9:A](eq A x_SupR_9 (multiply a (inverse a)
                               (multiply (inverse (inverse a))
                               (multiply a (inverse a) (inverse (inverse a))) (inverse a))))
                (H a (inverse a) (inverse (inverse a)) (multiply a (inverse a)
                  (inverse (inverse a))) (inverse a))
                (multiply a (inverse a) (inverse (inverse a)))
                (H2 (multiply a (inverse a) (inverse (inverse a)))
                  (multiply a (inverse a) (inverse a))))
              a
              (H4 a (inverse a)))
            a
            (H4 a (inverse a))))
        (inverse a)
        (H1 (inverse a) a))
      (inverse (inverse a))
      (H4 (inverse (inverse a)) a))))
```

Figura 6.4: Dimostrazione di `BOOO01-1` fornita da `auto paramodulation`

Capitolo 7

Integrazione con Matita

La tattica `auto paramodulation` è stata progettata ed implementata tenendo sempre in considerazione che il suo ambito di applicazione sarebbe stato il proof-assistant Matita. Concretamente, ciò significa che sono stati riusati per quanto possibile il codice e le strutture di dati di HELM, e l'interfaccia di invocazione è quella "standard" descritta in [Gal02]. L'integrazione di `auto paramodulation` in Matita, quindi, è seguita quasi totalmente da queste scelte iniziali. Gli unici due punti delicati sono stati il collegamento con la libreria di HELM, per sfruttare tutta l'informazione già codificata per le dimostrazioni, e la scelta dell'interfaccia utente da fornire per l'invocazione della tattica. Le due Sezioni di questo Capitolo sono dedicate pertanto ad illustrare e giustificare le scelte operate in questi due ambiti.

7.1 Collegamento con la libreria di HELM

Il primo problema che abbiamo dovuto risolvere per integrare il nostro lavoro in Matita ha riguardato l'utilizzo della libreria di teoremi di HELM. Durante il suo sviluppo, `auto paramodulation` non utilizzava infatti la libreria, ma per dimostrare un goal si serviva solamente delle ipotesi nel contesto locale, cioè quelle fornite dall'utente. Questo approccio chiaramente non è accettabile all'infuori di un ambito di sviluppo/test del codice, sia in quanto uno degli obiettivi principali del progetto HELM è quello di riutilizzare il più possibile la conoscenza già formalizzata e presente all'interno della sua libreria, sia perché tra le ragioni

stesse dello sviluppo di tattiche di dimostrazione automatica c'è quella di evitare all'utente di dover cercare egli stesso i teoremi utili alla dimostrazione di un goal.

È stato quindi necessario sviluppare un metodo di accesso alla libreria, che tenesse conto di due esigenze contrastanti:

1. innanzitutto, se un goal g è dimostrabile da `auto paramodulation` senza l'utilizzo della libreria, ma avendo nel contesto $H_1, \ldots, H_n$, g dovrebbe essere dimostrabile anche utilizzando la libreria, con un contesto vuoto e con $H_1, \ldots, H_n$ presenti nella libreria;

2. inoltre, se per la dimostrazione di un goal g non è necessario un teorema T presente in libreria, per questioni di efficienza esso non dovrebbe mai essere preso in considerazione da `auto paramodulation` nella ricerca della dimostrazione.

Naturalmente, i due requisiti appena esposti sono requisiti *ideali*: infatti, soddisfarli entrambi significherebbe conoscere in anticipo *tutti e soli* i teoremi necessari a dimostrare il goal, che equivale di fatto a conoscerne già una dimostrazione. È altrettanto ovvio che presi singolarmente i due punti sono banalmente soddisfacibili: per il primo, infatti, basterebbe utilizzare l'intera libreria - che contiene più di 30.000 elementi! - nella dimostrazione di qualsiasi goal, mentre per il secondo sarebbe sufficiente non considerare affatto la libreria. Chiaramente, nessuno di questi due metodi è particolarmente interessante: le soluzioni reali devono essere un compromesso tra completezza (punto 1) ed efficienza (punto 2).

La soluzione che abbiamo adottato è simile a quella utilizzata dalla tattica `auto`, e descritta in dettaglio in [Sel04]. Essa si basa sui metadati associati ad ogni elemento della libreria per identificare i teoremi e le definizioni "compatibili" con un certo goal. Più precisamente, vengono considerati nella costruzione della dimostrazione per un certo goal tutti gli elementi della libreria che

- sono equazioni, cioè hanno `cic:/Coq/Init/Logic/eq.ind#xpointer(1/1)` come costante in MainConclusion;

- soddisfano il vincolo *exactly* [Sel04] per l'insieme di costanti presenti nel goal, nel contesto locale e nei tipi e costruttori di tipi induttivi del goal

```
let equations_for_goal ~(dbd:Mysql.dbd) ((proof, goal) as status) =
  let _, metasenv, _, _ = proof in
  let _, context, ty = CicUtil.lookup_meta goal metasenv in
  let main, sig_constants = Constr.signature_of ty in
  match main with
  | None -> raise Goal_is_not_an_equation
  | Some (m, l) ->
      if m == UriManager.uri_of_string HelmLibraryObjects.Logic.eq_XURI
      then
        let set = signature_of_hypothesis context in
        let set = Constr.UriManagerSet.union set sig_constants in
        let set = close_with_types set metasenv context in
        let set = close_with_constructors set metasenv context in
        let set =
          List.fold_right Constr.UriManagerSet.remove (m::l) set in
        let uris =
          sigmatch ~dbd ~facts:false ~where:`Statement (main, set) in
        let uris = List.filter nonvar (List.map snd uris) in
        let uris = List.filter Hashtbl_equiv.not_a_duplicate uris in
        uris
      else raise Goal_is_not_an_equation
```

Figura 7.1: Codice della funzione `MetadataQuery.equations_for_goal`

e degli oggetti nel contesto. Se chiamiamo S tale insieme di costanti, un oggetto t della libreria soddisfa un vincolo *exactly* su S se

$$\exists S' \in \wp(S) \text{ t.c. constants\_of}(t) = S',$$

dove $\wp(S)$ è l'insieme potenza di S, e constants_of(t) è l'insieme delle costanti presenti in t.

In Figura 7.1 è riportato il codice della funzione `MetadataQuery.equations_for_goal` che implementa la procedura di ricerca nella libreria appena descritta. Essa viene invocata all'inizio dell'esecuzione di `auto paramodulation`, per determinare tutte le equazioni in libreria utili alla dimostrazione del goal. Prima di essere utilizzato, l'insieme viene filtrato per rimuovere i termini non trattabili da `auto paramodulation` - quelli cioè che non soddisfano la Definizione 4.1 - e quelli ridondanti.

Esempio 7.1. *Supponiamo di voler dimostrare il goal*

$$g \equiv \forall n : nat.\ (S\ n) * (S\ n) = (S\ (n + n + n * n)),$$

che ha tipo $\Pi n : nat.\ (S\ n) * (S\ n) = (S\ (n + n + n * n))$*. L'insieme delle costanti di* g *è*

$$\text{constants\_of}(g) = \{S, +, *\}$$

Considerando anche il contesto, che in questo caso contiene solo la dichiarazione $n : nat$ *ed i tipi ed i loro costruttori che compaiono nel goal, l'insieme delle costanti diventa:*

$$\{O, S, nat, *, +\}.$$

I teoremi e definizioni nella libreria che soddisfano il vincolo exactly *per questo insieme sono*[1]*:*

eq_add_S,	*mult_assoc*,	*mult_assoc_reverse*,
mult_comm,	*plus_assoc*,	*plus_assoc_reverse*,
plus_comm,	*plus_permute*,	*plus_permute_2_in_4*,
plus_reg_l,	*mult_0_l*,	*mult_0_r*,
mult_n_O,	*plus_0_l*,	*plus_0_r*,
plus_n_O,	*plus_Snm_nSm*,	*plus_Sn_m*,
plus_n_Sm,	*mult_plus_distr_l*,	*mult_plus_distr_r*,
mult_1_l,	*mult_1_r*,	*ZL0*,
S_to_plus_one,	*mult_n_Sm*.	

Il filtraggio per l'ammissibilità elimina eq_add_S e plus_reg_l, in quanto di tipo

$$\Pi\, n : nat.\Pi\, m : nat.(S\ n) =_{nat} (S\ m) \rightarrow n =_{nat} m$$

e

$$\Pi\, n : nat.\Pi\, m : nat.\Pi\, p : nat.p + n =_{nat} p + m \rightarrow n =_{nat} m$$

rispettivamente. Quindi, dopo l'eliminazione della ridondanza, l'insieme di

[1]per semplicità di notazione, sono riportati i nomi dei teoremi e non i loro `uri` completi.

teoremi applicabili diventa:

mult_assoc,	*mult_comm*,	*plus_assoc*,
plus_comm,	*plus_permute*,	*plus_permute_2_in_4*,
mult_0_r,	*plus_0_r*,	*plus_Snm_nSm*,
mult_plus_distr_l,	*mult_plus_distr_r*,	*mult_1_l*,
mult_1_r,	*mult_n_Sm*.	

Essi vengono trasformati in clausole positive e aggiunti a passive*; da quel momento l'esecuzione poi procede come se essi fossero ipotesi nel contesto locale.*

7.2 Interfaccia di invocazione da Matita

L'ultima operazione compiuta per integrare `auto paramodulation` in Matita è stata la definizione del modo in cui la tattica viene invocata dall'utente.

Inizialmente, si era pensato di invocare `auto paramodulation` dall'interno di `auto`, nel caso di goal equazionali. La motivazione di questa scelta risiedeva nel fatto che, come illustrato al Capitolo 4 (§4.3.2), `auto` è di almeno due ordini di grandezza più lenta di `auto paramodulation`. Tuttavia, al momento quest'ultima è di limitata applicabilità, trattando soltanto teorie puramente equazionali. Ciò significa che non solo non è in grado di dimostrare i goal che non sono equazioni, ma soprattutto che se un goal equazionale necessita per la sua dimostrazione di un teorema che non è un'equazione, `auto paramodulation` fallisce. In più, il costo di questo fallimento non è in generale trascurabile: in questi casi, pertanto, tentare di applicare `auto paramodulation` e quindi, in caso di fallimento, `auto`, causerebbe un peggioramento delle prestazioni.

Esempio 7.2. *Consideriamo, come semplice esempio di quanto appena illustrato, di voler dimostrare il goal*

$$\forall x : A.(f\ x) = n$$

nel contesto

$$\begin{aligned} A &: \textsc{Set} \\ f &: A \to A \\ n &: A \\ H &: \forall P : \textsc{Prop}.P \end{aligned}$$

Chiaramente, l'ipotesi H permette di concludere immediatamente la dimostrazione, ed in effetti `auto` *termina immediatamente con successo. Tuttavia, poiché H non è un'equazione,* `auto paramodulation` *non riesce ad utilizzarla, e quindi fallisce.*

Pertanto si è optato per una separazione delle due tattiche: la paramodulazione è attivata solo su richiesta dell'utente, passando il parametro opzionale *paramodulation* nell'invocazione di `auto`. Se il goal non è un'equazione, il parametro *paramodulation* viene ignorato ed `auto` viene invocata normalmente. Tuttavia, se il goal è un'equazione e viene passato il parametro *paramodulation*, se la tattica fallisce non viene invocata `auto`: la scelta di quale delle due tattiche automatiche utilizzare, infatti, viene lasciata all'utente di Matita.

Il problema dei parametri

Un problema ancora aperto in merito all'interfaccia utente offerta dalla tattica riguarda il settaggio dei parametri che governano la selezione delle clausole, l'ordinamento tra termini e la strategia di semplificazione utilizzata. Come abbiamo visto al Capitolo 4 (§4.3.1), questi parametri hanno un notevole impatto sulle prestazioni. Allo stesso tempo, i test effettuati dimostrano che il valore migliore di ciascuno di essi dipende sia dal goal in esame che dai valori degli altri. Tuttavia, il significato di ciascuno dei parametri, ed i modi in cui essi interagiscono, è chiaro solamente a chi ha una conoscenza abbastanza approfondita dell'architettura e dell'implementazione di `auto paramodulation`, e certamente tale conoscenza non è richiesta ad un generico utente di Matita.

Queste considerazioni ci hanno portato a porci un duplice problema, ovvero se consentire o meno di modificare i valori dei parametri, e - in caso affermativo - come permettere all'utente di farlo. Come già detto in precedenza, il problema è ancora aperto: al momento, infatti, non è possibile modificare i parametri quando `auto paramodulation` è invocata da Matita[2], ma la scelta non è ancora definitiva.

[2]tuttavia, abbiamo implementato anche una versione "stand-alone" della tattica, che consente di settare i parametri da riga di comando.

Capitolo 8

Conclusioni

Questa tesi è originata dall'esigenza di migliorare l'efficienza della tattica `auto` del proof-assistant Matita. Dopo una preliminare fase di studio del codice di `auto`, sono state sperimentate alcune ottimizzazioni su di esso, essenzialmente volte a limitare la dimensione dello spazio di ricerca su cui la tattica opera. Queste modifiche tuttavia, pur essendo efficaci, non hanno consentito miglioramenti sostanziali, ma hanno permesso di abbassare i tempi di esecuzione solo di un fattore costante (anche 3 o 4 nei casi migliori), mantenendo però invariato l'ordine di grandezza. In particolare, nel confronto con i moderni theorem-prover per la logica del primo ordine[1] (quali ad esempio Otter [McC03], Spass [Wei01], Vampire [Ria03], Waldmeister [HL02]) `auto` risultava più lenta di almeno un fattore 1.000, essendo quindi inutilizzabile per problemi non banali.

L'osservazione di questa enorme differenza di efficienza ci ha portato ad indagarne le ragioni, studiando l'architettura e l'implementazione dei theorem-prover più avanzati, Spass e Vampire in particolar modo. Abbiamo così verificato che la principale causa del divario di prestazioni è legata al predicato di uguaglianza, che in questi tool è trattato in modo speciale, tramite regole di inferenza dedicate, la principale delle quali prende il nome di *paramodulazione* [NR01].

Come naturale conseguenza di ciò, abbiamo deciso di sviluppare una nuova tattica per Matita, che si basasse anch'essa sulla paramodulazione, e che più in generale seguisse l'architettura di un theorem-prover automatico. Questa tesi è

[1] naturalmente, su problemi esprimibili in tale formalismo.

la descrizione dettagliata del prodotto di questo lavoro, la nuova tattica `auto paramodulation`.

Nonostante sia applicabile solo ad un sottoinsieme di problemi, ovvero solo a quelli puramente equazionali, `auto paramodulation` risulta essere molto efficace, riducendo i tempi di esecuzione di almeno due ordini di grandezza (come dimostrano i risultati presentati al Capitolo 4, §4.3).

8.1 Lavori correlati

Pur con le sue limitazioni, possiamo dire che `auto paramodulation` rappresenta un tentativo di integrazione tra dimostrazione automatica ed interattiva, cioè tra theorem-prover e proof-assistant. Un possibile scenario per il suo utilizzo è infatti la soluzione di problemi anche complessi, in cui la dimostrazione generale, la cui ricerca è guidata dall'utente, richiede la prova di alcuni sottogoal equazionali che possono essere trattati automaticamente invocando `auto paramodulation`.

Da questo punto di vista, il lavoro presentato in questa tesi ha molte analogie con analoghi tentativi di combinare i due approcci alla dimostrazione di teoremi, come ad esempio [Hur99], [Men03] e [ABH+98]. La differenza tra `auto paramodulation` ed i lavori citati sta nel fatto che in questi ultimi l'integrazione è ottenuta sviluppando un livello intermedio di comunicazione tra un proof-assistant ed un theorem-prover preesistenti e sostanzialmente indipendenti tra loro (HOL [GM93] e GANDALF [Tam97] nel primo caso, Isabelle [Pau94] e VAMPIRE nel secondo, e KIV [RSS97] e ${}_3TAP$ [BHOS96] nel terzo): in questi approcci, il problema viene tradotto dal linguaggio (solitamente di ordine superiore) del proof-assistant a quello (del primo ordine) del theorem-prover, che viene quindi invocato su questa traduzione del problema; infine, in caso di successo, deve essere operato il procedimento di traduzione in senso inverso per ottenere una prova valida per il proof-assistant. Al contrario, `auto paramodulation` ha un'architettura simile ad un theorem-prover, ma è a tutti gli effetti una parte di HELM, che ne condivide le strutture dati e gli algoritmi ed opera direttamente su termini CIC[2].

[2]pur accettandone, al momento, solo un sottoinsieme.

8.2 Sviluppi futuri

Abbiamo già sottolineato come `auto paramodulation` al momento si possa utilizzare solo per problemi puramente equazionali. Una naturale direzione di sviluppo è quindi l'estensione della tattica all'intero CIC, o perlomeno all'intera logica del primo ordine, con i connettivi standard $\neg$, $\wedge$ e $\vee$[3].

In particolare, un'estensione a tutta la logica del primo ordine potrebbe essere realizzata trasformando il goal ed i teoremi/ipotesi applicabili in forma a clausole, e quindi procedendo per risoluzione con paramodulazione come i tradizionali theorem-prover[4].

Un'estensione all'intero CIC, invece, richiederebbe un metodo diverso. Un approccio possibile consisterebbe nell'utilizzo di una tattica ibrida tra `auto` e `auto paramodulation`, in cui si alternino passi di riscrittura e semplificazione tramite paramodulazione e demodulazione, alla maniera di `auto paramodulation`, e passi di applicazione dei teoremi - sia quelli presenti in libreria e nel contesto che quelli generati dalle riscritture - alla maniera di `auto`.

In una diversa direzione, un altro possibile tema di sviluppo potrebbe essere l'utilizzo della tecnica di paramodulazione per generare automaticamente tutte le conseguenze "interessanti" di un insieme di teoremi e assiomi di partenza, per una qualche definizione di "interesse". Ad esempio, si potrebbe pensare di considerare interessanti tutti i teoremi usati molto frequentemente - come regole di riscrittura - nella dimostrazione di un certo goal, e quindi di proporre all'utente di inserire i più usati nella libreria.

Infine, l'implementazione attuale lascia ancora spazio a miglioramenti delle prestazioni tramite ottimizzazioni del codice.

[3] le cui definizioni in HELM sono rispettivamente `cic:/Coq/Init/Logic/not.con`, `cic:/Coq/Init/Logic/and.ind` e `cic:/Coq/Init/Logic/or.ind`

[4] bisogna tuttavia tenere conto del fatto che il metodo di risoluzione si applica a sistemi classici, mentre CIC, come tutti i formalismi basati sull'isomorfismo di Curry-Howard, è un sistema intuizionista.

8.2.1 Estensione all'intero CIC

Vogliamo terminare questa tesi con una descrizione un po' più dettagliata della direzione di sviluppo che attualmente sembra la più promettente: la possibilità, cioè, di risolvere goal costituiti da termini CIC arbitrari. Al momento attuale sia l'algoritmo che realizza la tattica estesa che il codice OCaml che lo implementa sono solo dei prototipi, tuttavia i risultati preliminari finora ottenuti appaiono promettenti.

La struttura generale della nuova tattica rimane quella descritta nei precedenti Capitoli - si utilizza cioè una variante dell'algoritmo *given-clause* per derivare nuove equazioni e per semplificare quelle esistenti - con la differenza che il goal può essere un termine CIC qualunque: in pratica ciò significa che la regola di superposizione sinistra non viene più utilizzata (in quanto il goal ora non viene più trattato come un'equazione, neanche nel caso in cui lo sia effettivamente), ma l'unica operazione effettuata sul goal è la semplificazione tramite demodulazione. Inoltre, oltre agli insiemi *passive* e *active* di equazioni, si utilizza un nuovo insieme, *theorems*, composto da teoremi in libreria e ipotesi locali che non sono equazioni. Ad ogni iterazione del *given-clause*, ciascun elemento di *theorems* viene applicato al goal corrente (chiamando la tattica primitiva `apply`), alla maniera di `auto`: se l'applicazione ha successo, il goal viene sostituito dal risultato della chiamata se questa genera dei sottogoal, altrimenti l'algoritmo termina con successo e si procede alla ricostruzione del termine di prova (come spiegato al Capitolo 6).

In Figura 8.1 è riportato lo pseudocodice di questa nuova versione di *given-clause*. La differenza principale rispetto al precedente sta nella condizione di terminazione con successo: poiché infatti ora *active* e *passive* contengono solo clausole positive, non è mai possibile derivare la clausola vuota[5], e l'algoritmo termina con successo solo grazie all'applicazione di un teorema al goal corrente. Questa nuova formulazione è più generale della precedente, ed infatti è possibile esprimere in essa anche la vecchia condizione di terminazione: basta semplicemente inizializzare l'insieme *theorems* al solo `refl_equal`, cioè la riflessività dell'uguaglianza. Essendo infatti esso applicabile solamente a termini

[5]ricordiamo che per noi la clausola vuota è in effetti una tautologia negativa, $a = a \rightarrow$.

var *goals, theorems:* insieme di termini
var *new, passive, active:* insiemi di clausole
var *current:* clausola
goals := {goal da dimostrare}
theorems := insieme dei teoremi applicabili al goal
active := $\emptyset$
passive := insieme delle clausole di input, tranne il goal
while *passive* $\neq \emptyset$ **do**
    goals := simplify_goals(goals, active, passive)
    theorems := simplify_theorems(theorems, active, passive)
    goals := apply_to_goals(theorems, goals)
    if *goals* = $\emptyset$ **then**
        return *"success"*
    end
    current := select(passive)
    passive := passive $\setminus$ {*current*}
    active := active $\cup$ {*current*}
    new := infer(current, active)
    new, active, passive := simplify(new, active, passive)
    passive := passive $\cup$ *new*
end
return *"failure"*

Figura 8.1: Pseudocodice per l'algoritmo *given-clause* della versione generale di `auto paramodulation`

del tipo `(eq A a a)`, ed essendo esso applicato solamente ai goal (quindi a termini "negativi"), l'algoritmo termina con successo - esattamente come nel caso precedente - solo quando si ottiene un goal che è una tautologia.

Gestione dell'insieme di goal

Il punto più delicato nella progettazione (e conseguente implementazione) di questa estensione sembra essere la gestione dell'insieme di goal aperti. Inizialmente, esso contiene un solo elemento, ovvero il teorema da dimostrare, tuttavia

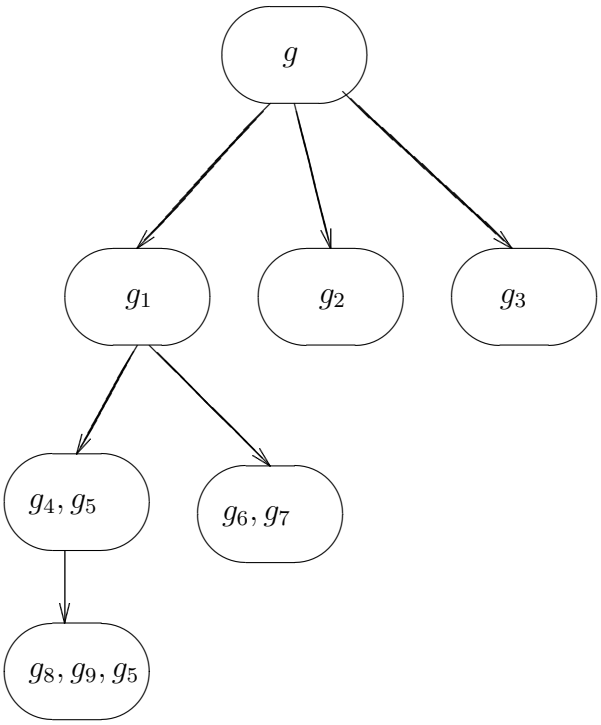

Figura 8.2: Esempio di albero AND-OR

ogni volta che l'applicazione di qualche teorema genera dei sottogoal, essi vengono aggiunti all'insieme. In effetti, da un punto di vista astratto l'insieme di goal è in realtà un albero "AND-OR" analogo a quello costruito da `auto` (§2.3.4): ogni nodo è composto da una lista in AND di (sotto)goal aperti, ed ha come figli i sottogoal generati dall'applicazione di diversi teoremi ad uno dei goal in AND.

Esempio 8.1. *Ad esempio, con l'insieme di teoremi*

$$g_1 \to g$$
$$g_2 \to g$$
$$g_3 \to g$$
$$g_4 \to g_5 \to g_1$$
$$g_6 \to g_7 \to g_1$$
$$g_8 \to g_9 \to g_4$$

ed il goal iniziale g, un possibile albero AND-OR è riportato in Figura 8.2.

Nella tattica `auto` questo albero viene attraversato (contemporaneamente alla sua costruzione) in maniera depth-first (fermandosi ad una certa profondità)

finché non si trova una soluzione oppure si esaurisce la visita. Questo approccio non può però essere adottato da `auto paramodulation`, in quanto ad ogni iterazione dell'algoritmo sia i goal che i teoremi applicabili vengono semplificati, rendendo così possibile che un teorema inapplicabile lo diventi proprio a causa di queste semplificazioni: ciò significa che ogni volta che avviene una semplificazione, si dovrebbe ricominciare la costruzione/visita dell'albero. È evidente però che questa soluzione è molto inefficiente. Al momento quindi stiamo sperimentando la seguente idea: l'insieme di goal è in effetti una lista (in OR) di liste di goal in AND; la lista è ordinata per profondità (intesa come profondità del nodo AND nell'albero AND-OR), con gli elementi più profondi che precedono quelli meno profondi; la funzione *apply_to_goals* scandisce la lista in OR tentando di applicare i teoremi disponibili al primo elemento della sottolista in AND: se il tentativo ha successo, tutti i risultati di queste applicazioni vengono aggiunti alla lista in OR, senza pero' togliere l'elemento da cui questi nuovi hanno avuto origine. Questo è indispensabile per poter fare "backtracking" qualora ciò fosse necessario.

Esempio 8.2. *Un semplice esempio permette di chiarire quanto appena esposto. Consideriamo nuovamente il goal g e l'insieme di teoremi introdotti nell'Esempio 8.1. Supponiamo che inizialmente gli unici teoremi a disposizione siano*

$$g_1 \to g$$
$$g_2 \to g.$$

La prima chiamata ad apply_to_goals*, effettuata sulla lista $[(0, [g])]$, restituirebbe $[(1, [g_1]); (0, [g])]$, la seconda $[(1, [g_1]); (1, [g_2]); (0, [g])]$. A questo punto, supponiamo che diventino disponibili anche i teoremi*

$$g_4 \to g_5 \to g_1$$
$$g_8 \to g_9 \to g_4.$$

Ora, la prossima chiamata ad apply_to_goals *esaminerebbe il primo elemento, $(1, [g_1])$, scoprendo che è possibile applicare ad esso il teorema $g_4 \to g_5 \to g_1$: la lista dei goal diventerebbe quindi $[(2, [g_4; g_5]); (1, [g_1]); (1, [g_2]); (0, [g])]$. A questo punto, quindi, procedendo ancora sul primo elemento, si vede che si può applicare $g_8 \to g_9 \to g_4$ al sottogoal g_4, ottenendo così la lista*

$$[(3, [g_8; g_9; g_5]); (2, [g_4; g_5]); (1, [g_1]); (1, [g_2]); (0, [g])].$$

Se, infine, dopo aver costruito questa lista, a seguito di alcune semplificazioni arrivassimo ad avere a disposizione ad esempio il teorema g_1*,* apply_to_goals *procederebbe nella scansione della lista fino al terzo elemento (facendo in effetti "backtracking"), applicando* g_1 *al quale si potrebbe concludere la dimostrazione.*

In conclusione, sottolineamo ancora come tutto ciò sia ancora in una fase decisamente primitiva: in particolare, non è ancora chiaro se la strategia di gestione dei sottogoal appena esposta funzioni in generale o se si presti a loop infiniti, in cui si continua ad applicare un teorema che genera altri sottogoal, ignorandone invece altri che potrebbero portare a concludere (anche se ciò potrebbe essere risolto con limitazioni alla profondità massima dell'albero o con euristiche che privilegino i teoremi che generano meno sottogoal), così come non è ancora stata affrontata la questione della completezza, ovvero se una strategia del genere porti sempre a trovare una dimostrazione se questa esiste. Tuttavia, gli esempi testati fino ad ora sono incoraggianti.

Bibliografia

[ABH+98] Wolfgang Ahrendt, Bernhard Beckert, Reiner Hähnle, Wolfram Menzel, Wolfgang Reif, Gerhard Schellhorn, and Peter H. Schmitt. Integration of automated and interactive theorem proving. In W. Bibel and P. Schmitt, editors, *Automated Deduction: A Basis for Applications*, volume II, chapter 4, pages 97–116. Kluwer, 1998.

[Bar92] H. Barendregdt. Lambda calculi with types. In Abramsky, Samson, et al., editors, *Handbook of Logic in Computer Science*. Oxford University Press, 1992.

[BG01] Leo Bachmair and Harald Ganzinger. Resolution theorem proving. In Alan Robinson and Andrei Voronkov, editors, *Handbook of automated reasoning*, volume I. Elsevier Science, 2001.

[BHOS96] Bernhard Beckert, Reiner Hähnle, Peter Oel, and Martin Sulzmann. The tableau-based theorem prover $_3$*tap*, version 4.0. In M. A. McRobbie and J. K. Slanley, editors, *Proc 13th CADE*, volume 1104 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[BS01] Franz Baader and Wayne Snyder. Unification theory. In Alan Robinson and Andrei Voronkov, editors, *Handbook of automated reasoning*. Elsevier Science, 2001.

[Coq04] Coq Development Team. *The Coq Proof Assistant Reference Manual, version 8.0*. INRIA, 2004.

[Dow01] Gilles Dowek. Higher-order unification and matching. In Alan Robinson and Andrei Voronkov, editors, *Handbook of automated reasoning*. Elsevier Science, 2001.

[Gal02] Michele Galatà. Sviluppo di tattiche per il supporto alla dimostrazione interattiva. Master's thesis, University of Bologna, 2002.

[GM93] M. J. C. Gordon and T. F. Melham. *Introduction to HOL (A theorem-proving environment for higher order logic)*. Cambridge University Press, 1993.

[GTL89] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Cambridge University Press, 1989.

[HL02] Thomas Hillenbrand and Bernd Löchner. A phytography of WALDMEISTER. *AI Communications*, 15(2-3):127–133, 2002.

[Hur99] Joe Hurd. Integrating Gandalf and HOL. In Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin, and Laurent Théry, editors, *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs '99*, volume 1690 of *Lecture Notes in Computer Science*, pages 311–321. Springer, September 1999.

[McC03] William McCune. OTTER 3.3 reference manual. Technical Report ANL/MCS-TM-263, Argonne National Laboratory, August 2003.

[Men03] Jia Meng. Integration of interactive and automatic provers. In Manuel Carro and Jesus Correas, editors, *Second CologNet Workshop on Implementation Technology for Computational Logic Systems*, September 2003.

[NR01] Robert Nieuwenhuis and Albert Rubio. Paramodulation-based theorem proving. In Alan Robinson and Andrei Voronkov, editors, *Handbook of automated reasoning*, volume I. Elsevier Science, 2001.

[Pau94] Lawrence C. Paulson. Isabelle: A generic theorem prover. *Lecture Notes in Computer Science*, 828, 1994.

[Ria03] Alexandre Riazanov. *Implementing an efficient theorem prover*. PhD thesis, University of Manchester, 2003.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.

[RSS97] W. Reif, G. Schellhorn, and K. Stenzel. Proving system correctness with KIV 3.0. In *14th International Conference on Automated Deduction. Proceedings*. Springer-Verlag, 1997.

[Sac04] Claudio Sacerdoti Coen. *Kowledge management of formal mathematics and interactive theorem proving*. PhD thesis, University of Bologna, 2004.

[Sel04] Matteo Selmi. Tattiche di dimostrazione automatica di teoremi su larghe basi di conoscenza. Master's thesis, University of Bologna, 2004.

[SRV01] R. Sekar, I. V. Ramakrishnan, and Andrei Voronkov. Term indexing. In Alan Robinson and Andrei Voronkov, editors, *Handbook of automated reasoning*, volume II. Elsevier Science, 2001.

[SS01] Geoff Sutcliffe and Christian Suttner. The TPTP problem library. TPTP v.2.4.1. Technical report, University of Miami, 2001.

[Tam97] T. Tammet. Gandalf. *Journal of Automated Reasoning*, 18(2):199–204, 1997.

[Wei01] Cristoph Weidenbach. *The theory of* SPASS *version 2.0*. MPI für Informatik, Saarbrücken, 2001.

[WRCS67] Lawrence Wos, George A. Robinson, Daniel F. Carson, and Leon Shalla. The concept of demodulation in theorem proving. *Journal of the ACM*, 14(4), 1967.