Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea in Informatica Magistrale

Tesi in Emerging Programming Paradigms

SVILUPPO DI UN INTERACTIVE THEOREM PROVER IN ELPI

Relatore:
Chiar.mo Prof.
Claudio Sacerdoti Coen

Presentata da:
Cristiano Piemontese

Sessione II
Anno Accademico 2017/18

So long, and thanks for all the fish

Indice

Introduzione

Tradizionalmente, i *proof assistant* sono stati implementati utilizzando linguaggi funzionali come OCaml (Coq, Matita) (Asperti, Ricciotti et al. 2011; Coq developement team 2017) o, piu raramente, imperativi come C++ (Lean) (de Moura et al. 2015). Queste implementazioni, sebbene rappresentino attualmente lo stato dell'arte, sono afflitte da alcune problematiche difficilmente risolvibili senza passare a linguaggi di programmazione di piu alto livello, come sostenuto in letteratura già da Felty (1989) e più recentemente da Guidi et al. (2018) e da Miller (2018). Tra queste problematiche ad esempio contiamo quelle derivanti dalla gestione di formule con *binders*, metavariabili e certi aspetti di unificazione di ordine superiore. Queste problematiche non sono dipendenti né da un particolare *prover* né tanto meno da una teoria. Per via della difficoltà di risoluzione di questi compiti si implementano algoritmi complessi. Altre difficoltà nascono dalla duplicazione di codice che si viene a creare tra *kernel* ed *elaborator*, le due componenti principali di un itp: nonostante il secondo dovrebbe essere un'estensione del primo, spesso quello che si verifica è che il secondo viene reimplementato da capo diventando così una riscrittura del primo con le aggiunte necessarie. Infine, vi è la necessità di rendere l'elaboratore estensibile, la quale porta la maggior parte dei *proof assistant* a definire linguaggi ad-hoc che ne aumentano la complessità (Guidi et al. 2018).

Nel corso di questa trattazione, supereremo lo stato dell'arte adottando il linguaggio ELPI (Dunchev, Guidi et al. 2015), ossia un'implementazione di un'estensione con vincoli del linguaggio λ-Prolog – a sua volta un'estensione di Prolog a una logica di ordine superiore (Miller e Nadathur 1988). Fin da subito, questo linguaggio si è mostrato adatto per svolgere compiti di meta-programmazione, come quello di implementare un *(interactive) theorem prover* (Felty 1989; Felty e Miller 1988; Miller e Nadathur 2012). Infatti, l'utilizzo di λProlog permette di demandare al linguaggio di programmazione la gestione di *binders* e metavariabili del linguaggio oggetto facendoli coincidere con quelli del meta-linguaggio, tramite approcci noti rispettivamente come *Higher Order Abstract Syntax* – o *λ-tree syntax* – e *semi-shallow embedding* (Dunchev, Sacerdoti Coen et al. 2016; Guidi et al. 2018). Nel corso del tempo λProlog ha ricevuto varie implementazioni, tra le quali ELPI, la più recente, si configura come un interprete veloce, *embeddable* e che introduce nuove *features* come costrutti di *constraint programming* e meccanismi per controllare la semantica generativa propria dei linguaggi logici. Come già discusso da Guidi et al. (2018), questi meccanismi, uniti alle capacità già presenti in λProlog permettono, se non sempre di risolvere, perlomeno di alleviare le problematiche sopra esposte.

Questa tesi tratterà della progettazione e conseguente sviluppo di alcuni componenti di un *interactive theorem prover* con ELPI.

Nello specifico, verrà innanzitutto mostrata l'implementazione di un *elaborator* (o *refiner*), il componente dei *proof assitant* che si occupa della traduzione di termini dati in input dall'utente, detti *termini parziali*, in termini modificati – detti anche raffinati – che a loro volta possono richiedere ulteriore elaborazione. L'*elaborator* è un componente la cui implementazione dipende, in parte,

dalla logica o teoria dei tipi per la quale lo si sta realizzando e, nell'ambito di questa tesi, la teoria *target* sarà Minimalist Type Theory (Maietti 2009).

Successivamente verrà discussa l'implementazione del *main loop* di *interactive theorem proving*, che prevede la gestione di comandi utente come tinycals (Sacerdoti Coen et al. 2007), tatticali LCF e tattiche primitive e la più generale gestione dei goal da dimostrare, in un modo che sia il più possibile indipendente dalla teoria con cui si sta lavorando.

La tesi prende il via, nel corso del primo capitolo, con la presentazione di λProlog sia dal punto di vista logico che da quello operativo. Tramite riferimenti storici di dimostrazione automatica, con particolare attenzione all'*interactive theorem proving*, verrà presentato il concetto di dimostratore interattivo e quindi specificato cosa ci si aspetta da esso. Sulla base di ciò verranno analizzati tentativi di implementazioni analoghe a quella presentata in questa tesi e verrà discusso se λProlog nella sua forma nativa sia sufficiente a soddisfare i requisiti posti. Infine, sarà presentato ELPI e saranno mostrate le estensioni che propone rispetto a λProlog per poi esaminare se queste permettano di realizzare ciò che mancava.

Nel secondo capitolo, verrà introdotta *Minimalist Type Theory*, ponendo in particolare attenzione sulla ricadute per l'implementazione. Verrà poi descritto il *kernel* di MTT implementato da Fiori e Sacerdoti Coen (2018) e su cui è stato costruito quanto implementato. Infine, verrà illustrato il progetto sviluppato, di cui sarà discussa l'integrazione col lavoro precedente e le estensioni, ossia l'unificazione e il sistema di dimostrazione interattiva.

Per concludere, verranno trattati nello specifico gli aspetti positivi e quelli negativi della soluzione proposta, fornendo possibilmente alcuni spunti per future rielaborazioni.

Capitolo 1

Programmazione logica e *Interactive Theorem Proving*

1.1 Programmazione logica *higher order*

Il paradigma della programmazione logica si presenta come un'alternativa rispetto ai più noti, quello imperativo e quello funzionale, non solo per la sua natura dichiarativa costituita di sentenze, siano esse fatti o regole, ma soprattutto per l'idea fondamentale che queste sentenze siano clausole di una precisa logica formale. Quella su cui si basa Prolog (Sterling e Shapiro 1994), un frammento della la logica del prim'ordine, è espressa tramite formule che prendono il nome di *first order Horn clauses* e che possono essere rappresentate nel seguente modo (Miller e Nadathur 2012, p. 43):

$$G ::= \top \mid A \mid G \wedge G \mid G \vee G \mid \exists X.G$$
$$D ::= A \mid G \Rightarrow D \mid D \wedge D \mid \forall X.D$$

dove i termini G rappresentano ciò che viene definito *goal*, D le *program clauses* e A una qualsiasi formula atomica.

λProlog è nato in seguito a tentativi effettuati da Miller e Nadathur (1988) di estendere Prolog con costrutti di ordine superiore. I loro studi li hanno prima condotti a tentare di estendere le clausole di *Horn* dal prim'ordine all'ordine superiore, poi a rivedere completamente i fondamenti su cui si costruiva la logica di Prolog e porre come base le *higher order hereditary Harrop formulas*.

Il primo aspetto per cui λProlog estende Prolog consiste dunque nella generalizzazione delle clausole di Horn a:

$$G ::= \top \mid A \mid G \wedge G \mid G \vee G \mid \exists X.G \mid \forall x.G \mid D \Rightarrow G$$
$$D ::= A \mid D \wedge D \mid \exists x.G \mid \forall X.D \mid G \Rightarrow D$$

Si noti che la distinzione tra X e x non è casuale e permette di differenziare tra metavariabili del linguaggio – che, come esposto in seguito, saranno espresse in sintassi concreta tramite lettere maiuscole – e variabili semplici. Le metavariabili sono sempre semanticamente quantificate esistenzialmente, mentre le variabili semplici universalmente. Pertanto solamente le prime sono soggette a istanziazione durante le prove.

Rispetto alle clausole di Horn le formule di Harrop permettono di:

1. quantificare su variabili di qualsiasi tipo;

2. quantificare sia esistenzialmente che universalmente nel corpo di un *goal*. La prima quantificazione opera su metavariabili mentre la seconda su variabili semplici (è vero invece il duale per le clausole del programma);

3. assumere clausole aggiuntive nella prova di un goal (formula $D \Rightarrow G$).

Un'ulteriore estensione sta nell'integrare il λ-calcolo tipato semplice nella sintassi dei termini. In particolare, in λProlog è possibile scrivere sia λ-astrazioni che applicazioni – `(x\...)` e `(m n)` in sintassi concreta.

Tutto questo permette di arrivare a poter scrivere in modo naturale un programma come il seguente, una sorta di *Hello world* per λProlog:

```
1 kind tm, ty type.
2 type lam ty -> (ty -> ty) -> tm.
3 type app tm -> tm -> tm.
4 type arr ty -> ty -> ty.
5
6 type of tm -> ty -> o.
7 of (lam A M) (arr T S) :- pi x\ of x T => of (M x) S.
8 of (app M N) S :- of M (arr T S), of N T.
```

Con queste poche righe di codice è possibile definire sia una codifica del λ-calcolo tipato semplice che un algoritmo di la *type inference*:

- seguendo la tecnica di *HOAS* i *binding* delle λ-astrazioni vengono fatti coincidere con quelli di λProlog, pertanto la traduzione di un termine come $\lambda x : T.M$ diventa banalmente `lam T M` dove M è una λ-astrazione di λProlog;

- permettendo di aumentare a *run-time* i *goal* da dimostrare con ulteriori clausole (`of x T => of (M x) S` nell'esempio), si rende possibile una corrispondenza quasi uno a uno tra le regole di tipaggio del λ-calcolo

tipato semplice e i predicati definiti per catturarle. Basti confrontare le regole effettive con la sintassi concreta usata per esprimerle.

$$\frac{\Gamma, (x : \tau) \vdash M : \sigma}{\Gamma \vdash \lambda x : \tau . M : (\tau \to \sigma)} \qquad \frac{\Gamma \vdash M : (\tau \to \sigma) \qquad \Gamma \vdash N : \tau}{\Gamma \vdash MN : \sigma}$$

1.1.1 La semantica di ricerca per λProlog

Un aspetto particolarmente importante per un linguaggio come Prolog è quello riguardante la semantica dei connettivi logici che hanno un'interpretazione non solo in termini di dimostrazione, ossia $G_1 \wedge G_2$ è vero se G_1 è vero e G_2 è vero, ma anche in termini di operazioni di ricerca in uno spazio di soluzioni. È allora spontaneo chiedersi in che modo si possa estendere ai connettivi delle formule di Harrop la semantica operazionale di cui godono quelli di Prolog.

Questa semantica può essere trovata nelle regole di introduzione dei connettivi della logica intuizionista così come descritte da Dunchev, Sacerdoti Coen et al. (2016):

- per dimostrare $G_1 \wedge G_2$ si procede a dimostrare G_1 e G_2 separatamente,
- per dimostrare $G_1 \vee G_2$ si prova a dimostrare G_1 e in caso di fallimento si fa *backtracking* a G_2,
- per dimostrare $D \Rightarrow G$ si assume D come clausola e si dimostra G in un contesto aumentato con D,

- per dimostrare $\forall x.G$ si dimostra G in cui tutte le occorrenze di x sono state sostituite dall'occorenza di una variabile fresca y,
- per dimostrare $\exists X.G$ si dimostra G in cui le occorrenze di X sono sostituite da Y metavariabile fresca,
- per dimostrare un goal atomico A lo si unifica con la testa di una qualche clausola del programma, eventualmente aprendo ulteriori *goal*.

Per quanto riguarda invece le *program clauses* si ha che:

- assumere $D_1 \wedge D_2$ significa assumere D_1 e D_2,
- assumere $\forall X.D$ significa, in teoria, assumere una quantità infinita di copie di D, una per ogni metavariabile fresca Y sostituita per X,
- assumere $\exists x.D$ significa assumere D in cui ogni occorrenza di x è sostituita con l'occorrenza di una variabile fresca y,
- assumere $G \Rightarrow D$ significa assumere che D valga sotto l'ipotesi che G valga, pertanto G va aggiunto all'insieme dei goal da dimostrare.

1.1.2 Unificazione di ordine superiore in λProlog

Un aspetto che finora è stato ignorato è cosa accade all'unificazione quando si passa da un contesto di prim'ordine a uno di ordine superiore. Una delle caratteristiche chiave di Prolog è infatti la possibilità di risolvere le *query*

utente producendo un assegnamento per le metavariabili presenti, al fine di rendere equivalenti due termini: quello su cui viene effettuata la *query* e che conterrà metavariabili — interpretabili come domande – e quelli presenti nella *knowledge-base*. Ad esempio siano definiti i seguenti predicati:

```
1 age george 26.
2 age maria 30.
3 age luigi 44.
```

Una *query* utente potrebbe prendere la forma di `age X 30` equivalente a chiedere chi ha 30 anni. Per trovare il valore di X (λ)Prolog deve istanziare la metavariabile X tramite un processo che prende il nome di unificazione del prim'ordine, poiché le istanziazioni per le metavariabili non possono essere funzioni/λ-astrazioni.

Uno dei punti di forza di Prolog è che questo tipo di unificazione è sia decidibile che efficiente (Martelli e Montanari 1976; Paterson e Wegman 1978). Tuttavia, altrettanto non si può dire per l'unificazione di ordine superiore che nella sua variante generale risulta indecidibile (Goldfarb 1981; Huet 1973). Questo è il tipo di unificazione presente in λProlog.

Infatti, supponiamo di definire in λProlog il seguente predicato che specifica la mappa di una funzione su una lista:

```
1 type mapfun (A -> B) -> (list A) -> (list B) -> o.
2
3 mapfun _ [] [].
4 mapfun F [X|Xs] [Y|Ys] :- Y = F X, mapfun F Xs Ys.
```

e supponiamo poi di voler eseguire le seguenti *query*:

```
1 ?- mapfun (x\ g a x) [a, b] L.
2 ?- mapfun F [a, b] [g a a, g a b].
```

La prima *query*, come atteso, produce la lista `[a a, b b]` mentre la seconda scatena un problema di unificazione di ordine superiore. In particolare, nel tentare di unificare `Y = F X` in cui `F` è stato unificato con `x\g a x` e `X` con `a`, si incontra il problema `g a a = F a` che ha quattro soluzioni possibili e nessun *most general unifier*: (`x\g x x`), (`x\g a x`), (`x\g x a`) e (`x\g a a`). Chiaramente solo una di queste soluzioni è corretta anche per `g a b = F b`, ossia `x\g a x`, tuttavia non c'è modo, in generale, di determinarlo in anticipo. Un interprete o compilatore λProlog potrebbe teoricamente implementare un algoritmo euristico di ricerca basato su *backtracking*, ad esempio quello derivato da quello definito da Huet (1975), ma avere come meccanismo di calcolo principale del linguaggio una ricerca euristica renderebbe il sistema troppo imprevedibile per potervi lavorare. L'idea di utilizzare un linguaggio logico di ordine superiore dovrebbe quindi essere accantonata se non fosse per gli sforzi di Miller (1991a), il quale ha dimostrato che esiste un frammento, che prende il nome di L_λ o *pattern fragment*, nel quale l'unificazione di ordine superiore è decidibile e ammette un *mgu*.

Il *pattern fragment*

Prima di introdurre il concetto di *pattern fragment* va presentato il contesto in cui l'unificazione avviene in λProlog. Un problema di unificazione può essere visto in chiave logica come l'asserzione di una congiunzione di uguaglianze sotto un prefisso costituito da quantificazioni su variabili. Per esempio, i problemi

di unificazione presenti in Prolog possono essere rappresentati come

$$\exists X_0, \ldots, \exists X_n.[t_0 = s_0 \wedge \cdots \wedge t_m = s_m]$$

dove le metavariabili $X_0, \ldots, X_n$ sono quantificate esistenzialmente in modo implicito. Questo schema, descritto da Miller (1992), può essere esteso a ugualianze sotto prefissi misti, ossia nella forma

$$\mathcal{Q}_0 X_0, \ldots, \mathcal{Q}_n X_n.[t_0 = s_0 \wedge \cdots \wedge t_m = s_m]$$

in cui i $\mathcal{Q}_i$ sono $\forall$ o $\exists$ e i t_i/s_i sono termini del λ-calcolo tipato semplice. In questo scenario Miller (ibid.) ha identificato una particolare classe di prefissi della forma $\forall\exists\forall$ che hanno un'interpretazione in termini di dichiarazioni di costanti e del loro *scope* rispetto alle metavariabili.

$$\forall x_0, \ldots, x_m.\exists X_0, \ldots, X_n.\forall y_0, \ldots, y_o.[t_0 = s_0 \wedge \cdots \wedge t_p = s_p]$$

In un prefisso in questa forma le quantificazioni assumono una semantica ben specifica:

1. il primo gruppo di quantificazioni universali può essere visto come le costanti del problema, che possono comparire nelle metavariabili,
2. il secondo gruppo rappresenta le metavariabili che vanno istanziate,
3. il terzo gruppo rappresenta variabili dichiarate nello scope delle metavariabili e che non possono comparire in esse.

Inoltre è possibile mettere in relazione un λ-termine con un problema di unificazione, ossia $\lambda x.M = \lambda x.N$ sse $\forall x.M = N$, elemento che costituisce il metodo principale per andare in ricorsione sui λ-termini, come visto nell'esempio in cui si è definita la *type inference* per il λ-calcolo tipato semplice.

Un problema di unificazione è nel frammento L_λ sse tutte le metavariabili che compaiono sono applicate a variabili distinte e che sono dichiarate nel loro scope di definizione, ossia un problema della forma

$$\forall x_0, \ldots, x_m.\exists X.\forall y_0, \ldots, y_n.[X\ y_0\ y_1\ \ldots\ y_o = t]$$

dove $o \leq n$ – non è necessario che X sia applicata a tutte le variabili.

Un esempio concreto è dato dai seguenti problemi:

$$\forall x.\exists X.X = x \qquad \exists X.\forall x.X = x$$

In questo contesto il primo problema ha una soluzione, ossia porre X a x. Il secondo non ha soluzione poiché non è possibile trovare un'istanziazione di X che vada bene per ogni x dal momento che x è dichiarato successivamente a X e non può comparire nel suo corpo. Per contro si consideri invece

$$\forall x.\exists X.(Xx) = x \qquad \exists X.\forall x.(Xx) = x$$

Il primo esempio chiaramente non è nel frammento, poiché X è applicato a una variabile che non è dichiarata nel suo *scope*. Infatti, questo problema ha due

soluzioni: $X = \lambda y.y$ e $X = \lambda y.x$. Viceversa il secondo problema rispetta la restrizione del frammento e, considerato che X non può usare x nel suo corpo, ha un'unica soluzione: $X = \lambda y.y$.

In λProlog ogni metavariabile è implicitamente considerata come se fosse quantificata esistenzialmente – come in Prolog – mentre ogni costante è considerata come se fosse quantificata universalmente prima di ogni metavariabile (è comunque possibile fornire una quantificazione esplicita). I problemi appena visti possono essere tradotti in sintassi concreta in questo modo:

```
1  pi x\ sigma X\ X = x.        % soluzione: X = x
2  pi x\ X = x.                 %            nessuna
3  pi x\ sigma X\ (X x) = x.    %            nessuna unica
4  pi x\ (X x) = x.             %            X = (y\ y)
```

Senza entrare nei dettagli dell'algoritmo di unificazione per il frammento, è possibile comunque dare un'idea generale di come avviene. Supponiamo che la metavariabile X vada unificata con un certo termine t. Questo termine potrà contenere sia costanti che variabili quantificate universalmente nello *scope* di una metavariabile, come ad esempio `pi x y\X x y = g x y`. Se nel termine da unificare compare una costante allora quella parte del termine potrà essere ottenuta solamente tramite mimica, mentre se nel termine compare una variabile "locale" allora o può essere ottenuta tramite proiezione di una delle variabili a cui X è applicata, oppure non vi è soluzione al problema.

Si consideri ad esempio i seguenti problemi

```
1  pi x y\ X x y = g x y.   % X = (m\ n\ g m n)
2  pi x y\ X x = g x y.     % nessuna soluzione
```

Da L_λ all'unificazione generale

Nonostante quanto descritto possa sembrare limitante, va considerato che è possibile scrivere numerosi programmi che non violano la restrizione del frammento L_λ. Ad esempio, tutto il codice del progetto svolto, cosi come quello di molti altri progetti scritti in λProlog/ELPI, rientra completamente in questo frammento. Qualora fosse necessario, peraltro, sarebbe possibile codificare con regole ad-hoc problemi di unificazione generali in modo tale da poter enumerare le loro soluzioni come soluzioni uniche di problemi nel frammento. Un esempio di questa codifica è rappresentata dalla coppia di predicati `copy` e `subst` definiti da Miller (1991b) per risolvere problemi di unificazione per termini chiusi del λ-calcolo tipato semplice:

```
1 copy (lam T F) (lam S G) :- copy T S, pi x\ copy (F x) (G x).
2 copy (app M N) (app P Q) :- copy M P, copy N Q.
3 copy X X.
4
5 subst F X Y :- pi x\ copy x X => copy (F x) Y.
```

Questo predicato ha due interpretazioni duali: la prima, come suggerisce il nome, è quella di codifica della sostituzione di un termine `X` nella λ-astrazione `F`; la seconda è invece quella di trovare quale λ-astrazione `F` in cui si è sostituito `X` ha prodotto `Y`.

Come sarà evidente più avanti nella descrizione dell'unificazione implementata nel progetto, la possibilità di effettuare queste codifiche permette di sfruttare l'algoritmo già presente in λProlog per ottenere algoritmi euristici che riescono a coprire una buona quantità di casi.

1.2 *Interactive theorem proving*

Come affermato da Harrison et al. (2014, p. 1):

> "per *interactive theorem proving* si intende una qualsiasi situazione in cui macchina ed essere umano lavorano assieme, in modo interattivo, per produrre una qualche dimostrazione formale"[1].

Nello specifico, la dimostrazione interattiva di teoremi si situa nel più generale ambito della dimostrazione automatica e riguarda tutte quelle applicazioni in cui l'essere umano ha un qualche tipo di intervento attivo. Si va da semplici *proof checkers*, i quali si occupano esclusivamente di verificare la correttezza formale del lavoro svolto (ad esempio il semplice programma mostrato in precedenza per il tipaggio del λ-calcolo tipato semplice), a veri e propri *proof assistant* che mirano ad aiutare lo svolgimento della prova attraverso un livello di automatismo che può essere più o meno elevato.

I primi studi riguardanti la dimostrazione automatica iniziarono verso la fine degli anni '60 e vennero effettuati nella direzione di una completa automatizzazione: l'intenzione era quella di riuscire a descrivere processi di inferenza che potessero sostituire completamente il ruolo dell'essere umano. Questi sforzi si concretizzarono in due direzioni principali: la prima, legata all'intelligenza artificiale, aveva come obiettivo quello di ricreare in una macchina i ragionamenti effettuati dagli esseri umani, tipicamente caratterizzati da un approccio di tipo *top-down* e basati su euristiche; la seconda, che vede le sue radici nella

[1] "By interactive theorem proving, we mean some arrangement where the machine and a human user work together interactively to produce a formal proof.", trad. mia.

logica matematica, mirava invece a usare le macchine in virtù della loro forza bruta e quindi senza particolare interesse a mimare un qualche tipo di processo cognitivo umano (MacKenzie 1995).

Per quanto riguarda la prima corrente, fu sicuramente importante il lavoro di Newell e Simon (1956), che vide la realizzazione di *Logic Theorist* (o *Logic Theory Machine*), considerato il primo dimostratore automatico. *Logic Theorist* fu creato per dimostrare teoremi presi da *Principia Mathematica* di Russel e Whitehead e venne dotato così di assiomi e regole di inferenza al fine di poter operare su di essi tramite un ragionamento *top-down*: partendo dal *goal* tentava di risalire al teorema da dimostrare applicando sia regole di inferenza che teoremi dimostrati in precedenza.

Nonostante *Logic Theorist* fosse riuscito a dimostrare una buona parte dei teoremi che gli vennero sottoposti, il lavoro di Newell e colleghi ricevette comunque numerose critiche da studiosi provenienti dall'ambito della logica matematica i quali lamentavano una notevole lentezza nella procedura di inferenza. Questa derivava da un problema di esplosione combinatoria che portava il dimostratore a dover considerare una quantità di *goal* esponenziale nel numero di regole di inferenza applicate, esplosione aggravata nel caso di *Logic Theorist*, che compiva passi di inferenza troppo piccoli. Lo sforzo per superare tale problematica portò a una procedura, ad opera di Robinson (1965), detta risoluzione, poi impiegata in Prolog, che riuscì ad alleviare il problema dell'esplosione combinatoria e generò un grande interesse. Tuttavia, verso la fine degli anni '70 l'attenzione fu posta altrove: presto ci si rese conto, anche grazie a risultati di teoria della complessità, che l'esplosione combinatoria non poteva essere risolta.

Tutto ciò portò a un abbandono di questo fronte di ricerca che si spostò,

soprattutto grazie alla sempre maggior importanza dell'informatica – che aveva una crescente necessità di verificare sia software che hardware — verso la dimostrazione assistita (MacKenzie 1995, pp. 11- 17)

1.2.1 Storia dell'*interactive theorem proving*

Tra i primi tentativi nella dimostrazione assistita sono sicuramente considerabili pionieristici quelli di Abrahams (1963) e soprattutto di Guard et al. (1969). Ma il primo *interactive theorem prover* che ebbe un impatto tale da iniziare quella che può essere considerata una tradizione di sistemi arrivata fino a oggi fu sicuramente *Automath* (Harrison et al. 2014, pp. 3-5).

Automath

Ad Automath (de Bruijn 1968, 1970). si deve innanzitutto l'utilizzo dell'isomorfismo di Curry-Howard per interpretare le dimostrazioni come termini di un λ-calcolo. In aggiunta a ciò, fu il primo progetto a seguire l'approccio che successivamente sarebbe stato formalizzato come *Logical Framework* (Harper et al. 1993), ossia quello di sviluppare un sistema in cui vi siano i minimi elementi necessari a codificare la logica di interesse. Inoltre, in onore di de Bruijn si deve l'omonimo Criterio di de Bruijn, definito da Barendregt (1997), che fissa l'idea che le dimostrazioni generate da un *proof assistant* debbano essere verificabili da un *proof checker* relativamente semplice e che a sua volta, almeno in principio, deve poter essere verificato manualmente (Asperti 2009; Harrison

et al. 2014).

Nei moderni *proof assistant* questo programma prende il nome di *kernel* e si configura come componente *trusted* da cui deriva la correttezza del sistema: prima, le dimostrazioni dell'utente vengono processate tramite un componente detto elaboratore; successivamente il risultato viene controllato dal *kernel* e accettato o scartato di conseguenza.

Edinburgh LCF

Di poco successivo ad Automath, Edinburgh LCF (Gordon et al. 1979) – successore di Stanford LCF – fu un dimostratore estremamente influente sia per quanto riguarda le tecniche di implementazione dei *proof assistant*, pensate per permettere di rispettare il criterio de Bruijn, sia per l'approccio all'estensibilità (Harrison et al. 2014, pp. 14-15). Infatti, in LCF i teoremi sono rappresentati tramite un tipo astratto `thm` che può essere manipolato solamente tramite regole di inferenza. Una dimostrazione diventa così una composizione di queste regole di inferenza e la correttezza è garantita per costruzione, nonostante non venga tenuta traccia di termini di prova. A Edinburgh LCF si deve anche la nascita del linguaggio di programmazione ML, da *Meta-Language*, concepito per essere un metalinguaggio tramite il quale implementare sia logiche che tattiche, attraverso le quali gli utenti possono estendere il sistema. Di fatto, queste sono delle funzioni, scritte in un linguaggio funzionale come ad esempio ML, che operano su oggetti di tipo `thm` descrivendo delle procedure di inferenza che concatenano passi base o altre tattiche. In quest'ultimo caso si parla di tatticali, che successivamente sono stati standardizzati di fatto in un insieme di tatticali detto "tatticali LCF".

1.2.2 Caratteristiche di un *interactive theorem prover*

A questo punto è possibile definire quelle che sono le caratteristiche principali che si richiedono a un *interactive theorem prover* basato sull'isomorfismo di Curry-Howard e di descrivere sinteticamente le parti di cui è composto.

L'isomorfismo di Curry-Howard

L'isomorfismo di Curry-Howard, detto anche corrispondenza di Curry-Howard o *formulae-as-types interpretation*, è una relazione tra programmi e dimostrazioni matematiche. Questa relazione deve il suo nome a Haskell B. Curry e William A. Howard, ai quali è attribuita la scoperta di questa corrispondenza (Curry 1934; Howard 1980), anche se è bene notare che essa trova radici già nell'interpretazione della logica intuizionista nota come *Brouwer-Heyting-Kolmogorov*. Idea base di questo isomorfismo è che vi sia una corrispondenza diretta tra il tipo dei termini dei λ-calcoli usati in teoria dei tipi e le formule di un sistema logico. Per estensione essa intercorre anche tra i tipi nei linguaggi di programmazione – che è possibile vedere come estensioni del λ-calcolo – e le formule di una logica. L'esempio più semplice che può essere dato è la relazione tra il λ-calcolo tipato semplice e la logica intuizionista minimale. Questa relazione emerge una volta che si definiscono le regole di tipaggio dei termini. Siano infatti termini e tipi definiti come

$$M ::= x \mid MM \mid \lambda x : T.M$$
$$T ::= \tau \mid T \to T$$

allora le regole di tipaggio sono le seguenti

$$\frac{\Gamma, (x : \tau) \vdash M : \sigma}{\Gamma \vdash \lambda x : \tau . M : (\tau \to \sigma)} \qquad \frac{\Gamma \vdash M : (\tau \to \sigma) \qquad \Gamma \vdash N : \tau}{\Gamma \vdash MN : \sigma}$$

e isolando i tipi si ottiene

$$\frac{\Gamma, \tau \vdash \sigma}{\Gamma \vdash \tau \to \sigma} \qquad \frac{\Gamma \vdash \tau \to \sigma \qquad \Gamma \vdash \tau}{\Gamma \vdash \sigma}$$

che non sono altro che le regole di introduzione ed eliminazione dell'implicazione per la logica proposizionale espresse in deduzione naturale. Vi è quindi non solo una corrispondenza tra tipo dei termini di un calcolo e formule di una logica ma anche tra il *type checking* di un termine e un albero di prova in deduzione naturale.

In generale, questa corrispondenza si riscontra a vari livelli: uno di questi permette di mettere in relazione la dimostrabilità di una formula logica con l'esistenza di un termine del tipo rappresentato da quella formula. Un *interactive theorem prover* alla Automath e quindi basato su questo isomorfismo è un dimostratore che, data in *input* una formula, permette di costruire un termine il cui tipo corrisponda a quella formula.

Un esempio di una tale derivazione potrebbe essere il seguente: sia data la formula della logica proposizionale $A \Rightarrow B \Rightarrow A$ e la si interpreti come il tipo di un'ipotetica funzione $F : A \to B \to A$. Si può notare che se F deve essere una funzione allora avrà forma $\lambda x : A.M$ dove a sua volta M sarà un termine del tipo $B \to A$ per le regole di tipaggio illustrate precedentemente. Ripetendo lo stesso ragionamento si ottiene che $M = \lambda y : B.N$ dove N deve avere tipo A. A questo punto N non può che essere x poiché è l'unico modo

per la funzione di ritornare un termine del tipo corretto. Quindi, un termine che dimostra $A \Rightarrow B \Rightarrow A$ è $\lambda x : A.\lambda y : B.x$. Va notato che data una formula esistono più termini che la realizzano, basti pensare a $A \Rightarrow A \Rightarrow A$.

Il kernel

Un *interactive theorem prover* creato secondo questo paradigma deve essere sicuramente costituito da un *proof checker* o *kernel* che si fa carico della correttezza dell'intero sistema. Infatti, il *kernel* deve implementare le regole di *type checking* – assieme a quelle di riduzione e conversione – del calcolo impiegato dal *prover* ed essere in grado di controllare che un termine dato in input da un utente sia ben tipato. Inoltre, per il criterio di de Bruijn il *kernel* deve avere una dimensione ridotta e non conterrà altro al di fuori di quanto menzionato.

L'Elaboratore

Se un dimostratore si limitasse a contenere il *kernel*, un utente non potrebbe fare altro che sottoporvi termini creati per soddisfare un tipo e sapere se questi sono o meno corretti, oppure chiedere di controllare che due termini/tipi siano convertibili, ossia uguali a meno di riduzione. Chiaramente questo modo di procedere non è quello desiderato, nonostante possa essere considerato comunque un metodo degenere di dimostrazione assistita. Ciò che tendenzialmente si chiede a un *proof assistant* è molto di più, ossia accompagnare l'utente durante la fase di creazione dei termini – e quindi di dimostrazione – e di alleviare quanto più possibile il compito di produrre una dimostrazione formale, che

può risultare da un lato tedioso per la quantità di dettagli che bisogna fornire e dall'altro, talvolta, perfino al di là delle capacità del singolo. Basti pensare a dimostrazioni che richiedono centinaia se non migliaia di pagine per essere sviluppate (List of long mathematical proofs 2018).

Di fatto,

> "[...] in pratica l'utente non interagisce mai col *kernel* e le rimanenti parti del sistema non dipendono dal comportamento del *kernel*. La vera intelligenza del sistema si trova nel secondo *layer*, chiamato elaboratore o *refiner*"[2] (Guidi et al. 2018, p. 2).

Questo componente ha come scopo principale quello di prendere in input termini utente e raffinarli, ossia trasformarli fino a far sì che siano effettivamente termini ben tipati che il *kernel* può accettare (Asperti, Ricciotti et al. 2012; de Moura et al. 2015; Guidi et al. 2018).

Fra i compiti dell'elaboratore possiamo contarne alcuni che portano l'uso di un *proof assistant* vicino a quello di un linguaggio di programmazione, come:

- ***Type inference***: spesso i termini possiedono annotazioni di tipo, come ad esempio potrebbe essere la definizione della funzione

  ```
  mapfun [a: type] [b: type] :: (a -> b) -> list a -> list b
  ```

  Per poter tipare questa funzione è necessario sapere chi `a` e `b` siano, tuttavia chiunque abbia mai usato un linguaggio di programmazione con *type inference* non ha mai dovuto specificare manualmente queste informa-

[2] "[...] in practice the user never interacts with the kernel and the remaining parts of the system do not depend on the behavior of the kernel. Where the real intelligence of the system lies is instead in the second layer, called elaborator or refiner", trad. mia

zioni (es. `mapfun int bool ...`). Un elaboratore deve offrire la stessa possibilità.

- **Risoluzione di *overloading***: in presenza di più definizioni per uno stesso termine l'elaboratore deve essere in grado di disambiguare automaticamente quale sia quello corretto al momento del suo uso.

- ***Coercion* implicita**: le omissioni lato utente possono arrivare perfino ad assumere l'esistenza di una qualche relazione di sotto-tipaggio implicito fra due tipi. Un esempio banale è utilizzare un numero naturale al posto di un valore booleano o viceversa. Chiaramente, questo è possibile solamente se l'elaboratore riesce a inserire automaticamente dei *cast* al tipo corretto. Questi possono essere o meno *user defined* e possono riguardare intere gerarchie di sotto-tipaggio.

Le possibilità di omissione non si limitano al livello dei tipi ma si estendono anche a quello dei termini. È possible che gli utenti lascino non specificati interi sottotermini che si aspettano possano essere generati automaticamente, come ad esempio

```
(_ :: X :: Xs) = [1, 2, 3, 4]
```

in cui `X = 2` e `Xs = [3, 4]`. Sebbene questo esempio sia piuttosto banale e corrisponda semplicemente a effettuare *pattern matching*, l'inferenza non è limitata solo a termini del prim'ordine ma anche a termini di ordine superiore, portando così, potenzialmente, a problemi di unificazione che, come è già stato discusso, sono indecidibili. Nonostante ciò, ogni dimostratore interattivo implementa una qualche forma di algoritmo di unificazione che, tramite una varietà di euristiche, tenta di inferire quanto più possibile e in modo da rispet-

tare ciò che l'utente si aspetta (tra molteplici soluzioni non tutte sono sensate e alcune sono preferibili) (de Moura et al. 2015; Ziliani e Sozeau 2015).

Nel corso della ricerca sulla dimostrazione assistita sono stati definiti anche una serie di concetti come *type classes*, *canonical structures*, *coercion pullbacks* e *unification hints* (Asperti, Ricciotti et al. 2009; Coen e Tassi 2008; Saibi 1999; Sozeau e Oury 2008) volti a tentare di alleviare il peso che grava sull'unificazione dando all'utente la possibilità di fornire dei modi "canonici" di risolvere certi problemi. Il caso più generale ed esplicito è sicuramente quello delle *unification hints*, che permettono di definire delle regole di unificazione della forma

$$\frac{\overrightarrow{?_x} \stackrel{?}{\equiv} \overrightarrow{H}}{P \stackrel{?}{\equiv} Q}$$

dove $\overrightarrow{?_x}$ sono *holes*, termini non istanziati presenti in P e Q per i quali viene definita una certa forma canonica di unificazione, ossia $\overrightarrow{H}$ (Asperti, Ricciotti et al. 2009). Un esempio di queste *hints* è dato in Asperti, Ricciotti et al. (ibid., p. 8):

$$\frac{?_1 := 0 \qquad ?_2 := 0}{?_1 + ?_2 \stackrel{?}{\equiv} 0}$$

che definisce la regola che, qualora ci si trovasse a dover unificare la somma di due metavariabili con 0 si potrebbe automaticamente concludere che la loro istanziazione è 0.

Un aspetto finora trascurato ma non più tralasciabile è quello della gestione dei termini aperti, ossia termini con *holes*, che non sono altro che variabili

quantificate esistenzialmente in modo implicito analoghe alle metavariabili di (λ)Prolog. Un esempio di questo di questo tipo di termine potrebbe essere una λ-astrazione in cui il corpo è lasciato implicito, rappresentabile come $\lambda x : \tau.?_f$ o $\lambda x : \tau.F$ (per rimanere fedeli alla sintassi della programmazione logica). Se, per l'isomorfismo di Curry-Howard, un termine ben tipato corrisponde a una dimostrazione, allora un termine aperto corrisponde a una dimostrazione aperta e ossia non ancora terminata, come definito da Geuvers e Jojgov (2002), i quali hanno mostrato come sia possibile estendere il tipaggio anche agli *holes*. Quest'ultimo, in particolare, va visto non tanto come giudizio che asserisce che una metavariabile **sia** di un certo tipo, quanto come un'affermazione che quella metavariabile **deve** essere di un certo tipo affinché tutto il termine sia considerabile ben tipato. Quello che si verifica, quindi, è che un elaboratore, dato un input come

$$\Gamma \vdash (\lambda x : \tau.F) : (\tau \to \sigma)$$

arriverebbe ad assumere che

$$\Gamma, (x : \tau) \vdash F : \sigma$$

dove quest'ultimo può essere interpretato come un *constraint* su F che va soddisfatto per poter istanziare il termine.

Pertanto, un elaboratore deve riuscire a gestire termini aperti e per ogni *hole* che vi compare deve anche memorizzare, in ciò che viene comunemente chiamato *metas-env*, i *constraint* di tipaggio che vigono su quella metavariabile. Ogniqualvolta una metavariabile viene istanziata, che sia in seguito a una

dimostrazione o in seguito alla richiesta dell'utente di unificare due termini, bisogna controllare che l'istanziazione rispetti effettivamente il *constraint*.

Infine, va menzionato il concetto di *constraint* canonico. Un *constraint* su una metavariabile è canonico se è in L_λ, vale a dire se la metavariabile compare applicata solamente a variabili distinte e dichiarate successivamente a essa. Questo si può vedere traducendo l'esempio precedente in sintassi concreta alla λProlog, con cui possiamo esprimere il *constraint* canonico

```
1 {..., x}: of x tau ?- of (F x) sigma.
```

dove '...' rappresenta eventuali altre variabili dichiarate e quindi presenti nel contesto ma che non riguardano `F`.

I *constraint* canonici sono particolarmente importanti perché vengono usati comunemente per assicurare l'unicità del tipaggio sulle metavariabili. Questo principio vale per i termini semplici e afferma, come si può immaginare, che ogni termine debba avere uno e un solo tipo. Per poter garantire che questo valga anche per gli *holes*, gli *interactive theorem provers* sono soliti mantenere l'invariante che su ogni *hole* vige un solo *constraint* e che questo è canonico. Questo invariante viene mantenuto in diversi modi: innanzitutto, alla creazione di un *hole* i *constraint* generati sono sempre canonici; poi, l'elaboratore non riduce un termine prima di aver effettuato il *type checking*, pertanto evita la perdita di eventuali *constraint* dovuta alla cancellazione di alcuni *hole*; infine, ogni volta che viene creato un *constraint* su una metavariabile si controlla se ne è già presente uno canonico ed eventualmente si combinano i due per ottenerne uno canonico (Guidi et al. 2018, p. 16).

Per quanto riguarda la dimostrazione vera e propria, essa viene effettuata tramite degli script in cui l'utente fornisce sequenze di passi fondamentali e applicazione di tattiche o tatticali tramite cui istanziare mano a mano i termini. A questo proposito vi sono in realtà diversi modi in cui un dimostratore può strutturare gli script di prova. Fino all'introduzione dei *Tinycals* da parte di Sacerdoti Coen et al. (2007) l'unico modo di poter svolgere una prova era attraverso l'uso di assiomi, come ad esempio le regole di introduzione ed eliminazione dei connettivi, oppure con l'applicazione di tattiche o tatticali LCF, con questi ultimi che rappresentano una sorta di insieme minimo di tatticali tramite i quali è possibile strutturare script complessi. I tatticali LCF sono i seguenti:

1. *id*: la tattica identità che non cambia lo stato.

2. *Then*: `t1 ; t2`, permette di comporre due tattiche. Viene eseguita la prima tattica, il suo output viene dato in input alla seconda e questa produce l'output finale.

3. *Orelse!*: `t1 | t2`, esegue la prima tattica e, in caso di fallimento, esegue la seconda. Questo tatticale si dice essere con *cut*: se la prima tattica ha successo, la seconda non verrà mai considerata come alternativa durante il *backtracking*.

4. *Try*: `try t` corrisponde a `t | id`, ossia prova a effettuare `t`.

5. *Solve*: `solve t` esegue `t` e ha successo solo se `t` non apre ulteriori *goal*.

6. *First*: `first L`, data una lista `L` di tattiche, le prova una a una e si ferma alla prima che ha successo.

7. *Repeat*: `repeat t` esegue `t` finché è possibile.

8. *Do*: `do n t` esegue `n` volte `t`.

9. *Branch*: `t ; [t_1 | ... | t_n ]`, esegue `t` - che ritorna n *goal* – e applica `t_i` a all'i-esimo.

Nonostante questi costrutti siano molto espressivi, presentano problemi dal punto di vista del *feedback* con l'utente. In particolare, durante l'esecuzione di uno script l'utente può visualizzare un cambiamento dello stato del sistema solamente tra uno *step* atomico e l'altro, dove uno *step* atomico è la più piccola unità di comandi che viene eseguita senza che vi si possa frapporre nulla. Quello che si verifica con i tatticali, in particolare `branch` e `;`, è che questo *step* atomico risulta eccessivamente grande e, spesso, lo stato pecedente all'esecuzione risulta troppo differente rispetto a quello successivo, facendo sì che l'utente non possa tenere traccia di cosa stia accadendo durante la chiamata e come si giunga al risultato ottenuto. Da questo punto di vista i *tinycals* rappresentano un tentativo di rimpicciolire la dimensione dello *step* andando a sostituire i tatticali con dei costrutti che permettono un controllo più fine. Fra questi costrutti, due specialmente vanno a sostituire i tatticali appena menzionati dando loro una nuova semantica:

- `branch` non è più un tatticale unico ma è separato in '`[`', ossia crea *branch*, '`|`', ossia *shift* o cambia *branch* e '`]`', ossia *unbranch* o chiusura del *branch*. Chiamare '`[`' permette di creare n *branch*, uno per ogni *goal* attualmente selezionato, che possono essere visualizzati ed elaborati separatamente. In un *branch* è possibile anche utilizzare il *tinycal* di *projection*, cioe '`i_j, ..., i_k:`' o '`*:`' che permette di selezionare un sott'insieme di *goal* su cui si è fatto *branch* e raggrupparli;

- ‘;’ non rappresenta più la composizione, che avviene “automaticamente”, ma è semplicemente un modo per distinguere un comando dall’altro. L’utilità deriva dal fatto che dopo ogni comando è possibile visualizzare lo stato ottenuto prima di scegliere il prossimo, operazione impossibile nel caso della composizione, che esegue direttamente la sequenza.

Gestione di libreria Come ultimo aspetto riguardante gli *interactive theorem prover* va sicuramente menzionata la necessità di organizzare definizioni e teoremi in una libreria. Chiaramente gli sforzi degli utenti andrebbero persi se, una volta terminata una dimostrazione lunga e complessa, non fosse possibile richiamarne il risultato per poterlo riutilizzare successivamente. Un dimostratore deve quindi essere in grado di memorizzare in un qualche modo queste informazioni per poterle sfruttare successivamente. Per ottenere questa funzionalità, qualunque dimostratore può semplicemente inserire definizioni e teoremi dell’utente in un insieme globale di ipotesi che ha a disposizione. Richiamare un risultato precedente equivale quindi a richiamare una di queste ipotesi.

Ciononostante va notato che vi sono esempi di gestione di libreria più sofisticati, com’è ad esempio quello di Matita, che indicizza i risultati in un vero e proprio motore di ricerca tramite il quale è possibile non solo trovare un qualsiasi risultato ma anche navigare le librerie grazie a *hyperlinks* tra oggetti indicizzati (Asperti, Guidi et al. 2004; Asperti, Ricciotti et al. 2011).

I giudizi

Prima di procedere ad analizzare come sia possibile implementare un dimostratore interattivo, è bene riassumere in modo un pò più formale quali sono i giudizi che vanno implementati per quanto riguarda *kernel* ed elaboratore. Per entrambi possiamo individuare cinque giudizi, di cui quelli dell'elaboratore dovrebbero essere un'estensione di quelli del *kernel*.

Sia $\Gamma ::= \epsilon \mid \Gamma, x : T$ un contesto, ossia una struttura che contiene asserzioni di tipaggio sulle variabili del nostro calcolo e siano gli input dei giudizi marcati come $\underline{i}$ mentre gli output come $\overline{o}$. Definiamo così i giudizi del *kernel*:

1. ***Type inference***: $\underline{\Gamma} \vdash \underline{t} : \overline{T}$, dato un contesto e un termine ritorna il tipo inferito. Questo corrisponde ai costrutti di tipaggio del calcolo implementato.

2. **Conversione**: $\underline{X_1} \equiv \underline{X_2}$, questo rappresenta l'ugualianza a meno di riduzione.

3. **Riduzione** $\underline{X_1} \to \overline{X_2}$, giudizio che corrisponde alla riduzione (sia tra termini che tra tipi) e corrisponde alle regole di $\beta\eta\ldots$-riduzione.

4. **Sostituzione**: $X\{t/x\}$, ossia dato un tipo/termine realizza la sostituzione di un termine t al posto di una variabile x.

5. ***Type checking***: $\underline{\Gamma} \vdash \underline{t} : \underline{T}$, dato un contesto, un termine e un tipo decide se il termine è del tipo specificato.

Per quanto riguarda l'elaboratore si ha una struttura addizionale, il *metas-env* menzionato in precedenza, che memorizza i *constraint* canonici sulle metavariabili: $\Sigma ::= \epsilon \mid \Sigma, (\Gamma \vdash X\vec{x} : T)$. Inoltre, in presenza di metavariabili, *type*

inference, *conversione* e *type checking* possono effettuare unificazione e pertanto produrre una sostituzione, indicata come Φ, ossia un assegnamento di termini a metavariabili che rispetti i *constraint* su di esse. I giudizi dell'elaboratore sono riassumibili come ($\rightsquigarrow$ indica la produzione di ulteriori output):

1. ***Type inference***: $\underline{\Sigma}, \underline{\Gamma} \vdash \underline{t} : \overline{T} \rightsquigarrow \Sigma', \Phi$, che può produrre una sostituzione e quindi modificare il *metas-env* Σ in Σ'.

2. ***Narrowing***: $\Sigma, \Gamma \vdash \underline{X_1} \equiv \underline{X_2} \rightsquigarrow \Sigma', \Phi$.

3. $\underline{X_1} \to \overline{X_2}$.

4. $X\{t/x\}$.

5. **Elaborazione**: $\Sigma, \Gamma \vdash \tilde{\underline{t}} : \underline{T} \rightsquigarrow \Sigma', \Phi, t'$, dato un termine $\tilde{t}$ — che potenzialmente contiene omissioni, abusi di notazione eccetera e quindi deve essere soggetto a elaborazione – si produce un termine t' raffinato, eventualmente modificando le metavariabili e tale per cui t' ha tipo T sono le ipotesi Γ, Σ'.

1.2.3 Implementare un *proof assistant* in λProlog

Avendo chiarito cosa si intende per *interactive theorem proving*, e di conseguenza, cosa ci si aspetta da un *proof assistant*, è lecito porsi la seguente domanda: λProlog è uno strumento adatto a realizzare tutto ciò? Per rispondere si farà riferimento a uno dei pochi lavori analoghi presenti in letteratura, ossia il dimostratore realizzato nella sua tesi di dottorato da Felty (1989), già introdotto

in parte in (Felty e Miller 1988). È a questo lavoro che si deve, soprattutto, di aver illustrato come sia possibile codificare una logica oggetto, o un calcolo, direttamente in λProlog e quindi come l'approccio *HOAS* possa facilitare questo lavoro. Particolarmente utile è anche da un lato la descrizione dell'architettura di un dimostratore a tattiche – con una componente interattiva – e dall'altro la codifica dei tatticali LCF in un linguaggio logico (Felty 1989, pp. 141-157).

Il seguente codice specifica tutti i giudizi presenti in un *kernel* per il λ-calcolo tipato semplice (ibid., p. 41-45):

```
1  kind ty, tm type.
2
3  type abs ty -> (tm -> tm) -> tm.
4  type app tm -> tm -> tm.
5
6  type '-->' ty -> ty -> ty.
7  type '#t'  tm -> ty -> o.
8
9  type redex tm -> tm -> o.
10 type red1  tm -> tm -> o.
11 type conv  tm -> tm -> o.
12
13 (abs M) #t (R --> S) :- pi X\ ((X #t R) => ((M X) #t S)).
14 (app M N) #t S :- M #t (R --> S), N #t R.
15
16 redex (app (abs M) N) (M N).
17 redex (abs X\ (app M X)) M.
18
19 red1 M N :- redex M N.
20 red1 (abs M) (abs N) :- pi X\ (red1 (M X) (N X)).
21 red1 (app M N) (app P N) :- red1 M P.
22 red1 (app M N) (app M P) :- red1 N P.
23
24 conv M N :- red1 M N.
25 conv M M.
```

```
26 conv M N :- conv N M.
27 conv M N :- conv M P, conv P N.
```

Il primo *statement* dichiara due nuovi tipi primitivi del meta-linguaggio `tm` e `ty`, utilizzati per rappresentare il tipo rispettivamente dei termini e dei tipi del linguaggio oggetto. Com'è già stato anticipato λProlog estende Prolog con i termini del λ-calcolo tipato semplice, dunque è possibile dichiarare nuovi meta-tipi con la *keyword* `kind`. Naturalmente, questi meta-tipi possono essere tipi semplici o funzioni. Nella sezione seguente viene dato il tipaggio dei costrutti utilizzati che codificano rispettivamente: λ-astrazione e applicazione; il tipo funzione; i giudizi di tipaggio, riduzione e conversione.

Questa soluzione mostra immediatamente il vantaggio di utilizzare un linguaggio come λProlog e ossia l'estrema compattezza del codice, dovuta in parte allo stile di programmazione logica ma soprattutto al fatto che il linguaggio sottostante implementa già tutta una serie di funzionalità, come la *capture-avoiding substitution*, che altrimenti richiederebbero innumerevoli righe di codice. Nonostante ciò, è incompleta per quanto riguarda gli scopi proposti: qualora si utilizzasse questo codice in presenza di metavariabili il comportamento ottenuto non sarebbe quello desiderato. Si consideri la *query* `?- (abs X) #t (nat --> nat)`. Il risultato ottenuto sarebbe una enumerazione di termini ben tipati generati da λProlog

```
1 X = x\ x.
2 X = x\ app (abs (y\ y)) x.
3 X = x\ app (abs (y\ y)) (app (abs (y\ y)) x).
4 ...
```

mentre il risultato desiderato sarebbe la creazione di un *constraint* del tipo `(X x) #t nat`. Questo problema deriva dalla semantica generativa utilizzata dai linguaggi logici.

Un ulteriore problema è dato dalle regole di riduzione che, di nuovo, assumono l'uso solamente di termini chiusi. Questo fa sì che una *query* come `?- redex (app (abs X) x) x.` risulti in un problema di unificazione esterno al frammento, corrispondente a

$$\forall x.\exists X.(Xx) = x$$

Per ovviare a entrambi questi problemi – e quindi poter creare un elaboratore – sarebbe necessario un meccanismo per poter riconoscere le metavariabili e definire un comportamento alternativo, meccanismo non presente in λProlog.

Per quanto riguarda il dimostratore interattivo, di particolare interesse è l'implementazione dei tatticali LCF (Felty 1989, p. 143):

```
1  then Tac1 Tac2 InGoal OutGoal :-
2      Tac1 InGoal MidGoal, maptac Tac2 MidGoal OutGoal.
3  orelse Tac1 Tac2 InGoal OutGoal :-
4      Tac1 InGoal OutGoal; Tac2 InGoal OutGoal.
5  idtac Goal Goal.
6  repeat Tac InGoal OutGoal :-
7      orelse (then Tac (repeat Tac)) idtac InGoal OutGoal.
8  try Tac InGoal OutGoal :- orelse Tac idtac InGoal OutGoal.
9  solve Tac InGoal tt :-
10     Tac InGoal OutGoal, goalreduce OutGoal tt.
```

Come si può osservare i tatticali hanno una struttura completamente naturale che corrisponde alla loro interpretazione: `then` effettua la composizione di due tattiche utilizzando `maptac`, ossia l'applicazione di una tattica a tutti i *goal* aperti, `orelse` è banalmente implementato tramite il '`;`' di λProlog e via dicendo. Questi tatticali vengono impiegati tramite il componente interattivo realizzato dal *main loop* di dimostrazione che prende in input una tattica dall'utente e prova ad applicarla. In caso di fallimento si effettua *backtracking*, in caso di successo si avanza lo stato della prova. È prevista anche la possibilità di terminare la dimostrazione preventivamente tramite `quit` e di annullare un comando dato in input tramite `backup` (Felty 1989, p. 150).

Sebbene il dimostratore realizzato mostri effettivamente capacità interattive, di nuovo va evidenziato che il risultato non è completamente soddisfacente, in parte rispetto al comportamento atteso e in parte dal punto di vista implementativo. Innanzitutto le regole di inferenza sono state completamente re-implementate come tattiche (ibid., pp. 138-139). Questo va in contrasto con la volontà di lasciare nel *kernel* solo i giudizi di tipaggio, vale a dire le regole di inferenza della logica. Di nuovo, non è possibile farlo a meno di operare in un contesto con *constraint*. Infatti, se si sostituissero le regole di inferenza alle tattiche e si facesse sì che le tattiche istanziassero solamente termini – ad esempio la tattica dell'introduzione dell'implicazione istanzia una λ-astrazione eccetera – si incorrerebbe nel problema descritto in precedenza e ossia che appena si chiede il tipaggio di un termine aperto questo termine viene istanziato automaticamente, impedendo di fatto la dimostrazione interattiva. In secondo luogo – ma questa è una scelta implementativa che può essere cambiata – il dimostratore implementato è *depth-first* e quindi obbliga l'utente a procedere con un ramo di dimostrazione fino a che non è concluso.

1.2.4 ELPI: λProlog rivisitato

Tutte le problematiche evidenziate finora rispetto a λProlog trovano una soluzione in ELPI, un interprete per λProlog sviluppato da Dunchev, Guidi et al. (2015) e che estende ulteriormente il linguaggio con il supporto per i *constraint* e con possibilità di controllo e limitazione della semantica generativa (Dunchev, Sacerdoti Coen et al. 2016; Guidi et al. 2018). Queste estensioni, come sostenuto da Guidi et al. (2018), dovrebbero rendere λProlog completamente adatto a implementare un dimostratore interattivo.

Semantica generativa e `mode`

La prima estensione è piuttosto diretta: alla definizione di un predicato, ad esempio `predicate A B C`, è possibile specificare come i vari argomenti di quel predicato vadano trattati ossia come input o come output. In sintassi concreta: `mode (predicate i i o)`, per l'esempio precedente. I modi indicano rispettivamente la seguente politica:

1. '`o`': la semantica è quella di default e quindi generativa. L'interprete può effettuare unificazione dell'argomento.

2. '`i`': la semantica è conservativa e quindi l'interprete non effettuerà nessuna unificazione del parametro formale con l'attuale, ma semplicemente il *pattern-matching* sull'argomento.

Questo costrutto è particolarmente utile per vari motivi. Il primo, una sorta di *side-effect* del design, è che permette di definire predicati con un comportamento più predicibile e simile a quello di una funzione nel paradigma funzionale.

Infatti, si tenderà a scrivere codice in cui sono individuati un insieme di input su cui viene effettuato *pattern-matching* e su cui si va in ricorsione. Il secondo motivo, quello inteso, è di poter scrivere codice come:

```
1  kind tm, ty type.
2  type lam ty -> (ty -> ty) -> tm.
3  type app tm -> tm -> tm.
4  type arr ty -> ty -> ty.
5
6  mode (of i o).
7  type of tm -> ty -> o.
8  of (lam A M) (arr T S) :- pi x\ of x T => of (M x) S.
9  of (app M N) S :- of M (arr T S), of N T.
10 of (uvar as X) T :- declare_constraint (of X T) [X].
```

Con questa piccola aggiunta (riga 6 e 10) si è esteso il comportamento del *type checker* a termini aperti: quando incontra una metavariabile – a cui viene dato il nome `X` in quel contesto – viene dichiarato un *constraint* di tipaggio su di essa e sospende la ricerca. In questo modo la *query* `?- of (lam nat X) (arr nat nat)` non genererà più una soluzione per `X` ma creerà un *constraint* della forma `of x nat ?- of (X x) nat`. Un eventuale assegnamento a `X` risveglierebbe questo *constraint* e la verifica di tipo verrebbe ripresa. Assegnare un termine sbagliato equivale al fallimento del tipaggio, che ritornerebbe a sospendersi tramite il *backtracking*.

Constraint, CHR ed elaborazione

Le seconda estensione riguarda la possibilità di manipolare i *constraint* dichiarati dall'utente tramite regole ispirate alle *Constraint Handling Rules* di

Frühwirth (1998). Queste regole permettono all'utente di definire modi per manipolare i *constraint* presenti, rappresentati sotto forma di predicati in CHR e sequenti in ELPI. Una presentazione dettagliata della semantica è consultabile in (Guidi et al. 2018, pp. 11-14), tuttavia il seguente esempio dovrebbe dare un'idea generale del loro funzionamento.

```
1 constraint of {
2     rule (of X T)
3     \    (of X T')
4     |    (not (T = T'))
5     <=>  ([] ?- fail).
6 }
```

Questa regola implementa un semplice controllo di unicità del tipaggio. Ogni `rule` è dichiarata in un blocco `constraint` che permette di specificare quali ipotesi si desidera recuperare dal contesto. In questo caso è stato scelto di recuperare solo le ipotesi `of`. Subito dopo `rule` è possibile specificare una lista di sequenti sulla quale si vuole fare *match* tra i *constraint* attualmente presenti. Dopo '`\`' è invece possibile indicare una lista di sequenti, ossia *constraint*, che si desidera rimuovere se l'esecuzione della regola ha successo. Nella parte '`|`' si può inserire codice che funge da guardia e determina il successo della regola. Infine, con '`<=>`' viene indicato il sequente da sostituire a quelli rimossi.

Per esempio, se dopo aver chiesto il tipaggio di `app (lam nat y\y) X` chiedessimo quello di `app (lam bool (y\y) X)`, la regola farebbe *match* ed eseguirebbe `not (nat = bool)`. Questo avrebbe successo, porterebbe alla sostituzione del sequente `[] ?- fail` e conseguentemente al fallimento, poiché la stessa metavariabile non può avere due tipi differenti nello stesso contesto. I sequenti usati per fare *match* presentano, in questo caso, solamente il corpo.

Sarebbe stato possibile recuperare il contesto tramite la sintassi `Ctx ?- of X T`: in `Ctx` si troverebbe la lista di ipotesi assunte fino a quel momento, ad esempio `Ctx = [of y nat]`.

Come si può intuire da questo esempio, le regole di gestione dei *constraint* presentano un grande potenziale. Per esempio, permettono una gestione arbitraria e sofisticata dei *constraint* definiti e possono anche essere aggiunte dall'utente per modellare conoscenza di dominio. Inoltre, come si avrà modo di vedere, consentono senza particolare sforzo di integrare la procedura di dimostrazione interattiva con il *kernel* senza che questa debba effettuare continue richieste di tipaggio, conversione e riduzione: basta lanciare il predicato di tipaggio e tenere semplicemente traccia delle metavariabili da istanziare, poiché i controlli avverranno automaticamente al risveglio dei *constraint* su di esse.

Capitolo 2

Un *interactive theorem prover* per *Minimalist Type Theory*

L'*interactive theorem prover* che verrà illustrato è stato creato con l'intenzione di permettere di effettuare dimostrazioni per la teoria dei tipi proposta da Maietti (2009), ossia *Minimalist Type Theory*, una teoria fondazionale per la matematica costruttivista volta a dare un *core* comune tra le teorie già esistenti in teoria degli insiemi, teoria delle categorie e teoria dei tipi.

Questa teoria ha la caratteristica innanzitutto di essere una teoria "alla Martin-Löf", vale a dire che è costituita da numerose regole, le quali definiscono meticolosamente: le relazioni tra tipi; le leggi che governano il tipaggio di termini; il loro uso tramite le regole di eliminazione; regole di conversione e riduzione a cui sono soggetti. Tutto ciò ha una ricaduta concreta sulla realizzazione di *kernel* ed elaboratore poiché, ovviamente, ne influenza la dimensione: più termini sono presenti, più casi vi saranno da gestire.

Un ulteriore aspetto da considerare è la presenza di tipi dipendenti, che complica notevolmente i *judgement*, la rappresentazione dei *goal* e l'elaborazione,

poiche rende il tipaggio/*type inference* dipendente dalla conversione/unificazione.

Un'ultima caratteristica, rilevante soprattutto per la realizzazione del *kernel* (Fiori e Sacerdoti Coen 2018), è la composizione in due livelli della teoria: uno intensionale e uno estensionale. La differenza tra questi due livelli, brevemente, è spiegabile in termini di gestione delle uguaglianze tra termini e, in particolare, rispetto alle regole definite per esse. In una teoria intensionale, la regola di eliminazione dell'uguaglianza necessita di aver costruito in precedenza un termine per il tipo dell'uguaglianza e, quindi, per l'isomorfismo di Curry-Howard, di essere in possesso di una dimostrazione. Invece, in una teoria estensionale, l'uguaglianza assume - senza specificare in che modo - di riuscire a costruire questa dimostrazione. Questo fa sì che il tipaggio diventi indecidibile, richiedendo di integrare una procedura di ricerca generale di prove che potrebbe non terminare mai.

2.1 Un *kernel* per *Minimalist Type Theory*

Il lavoro svolto si va a integrare con quanto fatto da Fiori e Sacerdoti Coen (ibid.), che hanno implementato i *type checker* sia per il livello intensionale che per quello estensionale, quest'ultimo in seguito a una riformulazione atta a rendere *syntax directed* alcune regole.

Prima di procedere a descrivere l'integrazione con questo lavoro è opportuno dare una descrizione del sistema precedente all'estensione. Per ulteriori dettagli

è possibile consultare Fiori e Sacerdoti Coen (2018).

Il sistema presenta un *main file*, ossia `main.elpi`, contenente:

1. le definizioni di base dei giudizi di tipaggio, conversione e riduzione, comuni a tutti i termini: `of`, `ofType`, `isa`, `isaType`, `locDecl`, `locTypeDef`, `conv`, `dconv`, `hnf` e `hstep`;

2. la definizione di predicati atti a gestire una libreria di dimostrazioni, contenente la codifica di meta-termini per rappresentare definizioni/teoremi/lemmi.

Le regole che implementano i giudizi per un tipo particolare sono definite in un file `nometipo.elpi` in una *directory* denominata `calc`. Ognuno di questi file si articola a sua volta come una sequenza di definizioni per i predicati sopra descritti, ciascuna, chiaramente, specializzata per il tipo verso cui si sta estendendo il predicato.

I giudizi riguardanti il tipaggio sono `of`, `ofType`, `isa`, `isaType`, `locDecl` e `locTypeDef`. I primi due definiscono la *type inference* rispettivamente dei termini e dei tipi: sono entrambi predicati ternari definiti con `mode (of i o o)` e `mode (ofType i o o)`, ossia prendono in input termine/tipo e ne ritornano tipo/*kind* e livello. Quest'ultimo parametro viene utilizzato per determinare per quale livello della teoria, estensionale o intensionale, vale il giudizio e permette il riutilizzo dei predicati enunciati. `isa` e `isaType` definiscono invece il *type checking* e sono descritti da regole comuni presenti in `main.elpi`:

```
1 mode (isa i i o).
2 mode (isaType i i o).
```

```
3
4  isaType Type Kind IE :-
5      ofType Type Kind' IE,
6      pts_leq  Kind' Kind.
7
8  isa Term TY IE :-
9      of Term TY' IE,
10     conv TY' TY.
```

Per quanto riguarda un tipo, è di un certo `Kind` se è inferibile che sia di un `Kind'` e fra questi intercorre una relazione rappresentata da `pts_leq`. Similmente, per i termini si inferisce il tipo del termine e si controlla che sia convertibile con quello dato in input.

Si può notare che entrambi i predicati sono definiti con un `mode` in cui tutti gli argomenti tranne l'ultimo sono input. Questo rappresenta la semantica corretta del giudizio di *type checking* (l'ultimo argomento è estraneo a esso).

Infine, gli ultimi due predicati codificano le dichiarazioni locali di termini (`locDecl`) e tipi (`locTypeDef`). Uno degli usi è quello di poter tipare variabili fresche introdotte tramite `pi` mentre si va in ricorsione su termini con *binders*. Un esempio è dato dalla regola di eliminazione del tipo Σ:

```
1  of (elim_setSigma Pair MTy M) (MTy Pair) IE  :-
2      isa Pair (setSigma B C) IE,
3      pi z\ locDecl z (setSigma B C) => ofType (MTy z) _ IE,
4      pi x y\ (locDecl x B, locDecl y (C x)) =>
5          isa (M x y) (MTy (pair B C x y)) IE.
```

In questa regola si può vedere l'uso di molti dei costrutti discussi. Va sottolineata la scelta implementativa di inserire annotazioni di tipo nei termini alla Church, che rende decidibili alcuni giudizi. Confrontando questa regola

con quella definita in Maietti (2009, p. 39), è evidente quanto poco complesso sia tradurre questo tipo di problemi in λProlog:

$$\frac{\begin{array}{ccc} & & M(z)\ col\ [z \in \Sigma_{x \in B} \in C(x)] \\ & & \vdots \\ b \in B \qquad c \in C(b) & & m(x,y) \in M(\langle x,y \rangle) \end{array}}{El_{\Sigma}(\langle b,c \rangle, m) = m(b,c) \in M(\langle b,c \rangle)}$$

Per quanto riguarda le regole di conversione e riduzione si hanno rispettivamente `conv`/`dconv` e `hnf`/`hstep`. Sempre in `main.elpi` sono definiti i casi comuni:

```
1  mode (hnf i o).
2  mode (hstep i o).
3  hnf A B :- hstep A C, !, hnf C B.
4  hnf A A.
5
6  mode (conv i i).
7  mode (dconv i i).
8  conv A A :- !.
9  conv A B :- locDecl _ (propEq _ A B), !.
10 conv A B :- locDecl _ (propEq _ B A), !.
11 conv A B :- hnf A A', hnf B B', dconv A' B'.
12
13 dconv A A :- !.
```

`conv` esprime l'uguaglianza estensionale tra due termini o tipi: il caso base (riga 9) è che ogni termine/tipo sia convertibile con sè stesso; oppure, due termini/tipi sono convertibili se è dichiarato un termine del tipo `propEq`, ossia l'equivalenza estensionale; infine, due termini/tipi sono convertibili se le loro *weak head-normal form* sono convertibili. L'uso di `dconv` serve per evitare di riconsiderare il caso della `propEq`: ogni termine definisce `dconv` che implementa

la conversione vera e propria, per poi eventualmente richiamare `conv` nel suo corpo.

`hnf` e `hstep` implementano, invece, i giudizi di riduzione. In particolare, il primo riguarda la normalizzazione a forma normale di testa debole, ossia riduce fintanto che il termine in testa è un *redex*. `hstep`, invece, è usato per i *redex* veri e propri.

Per avere un'idea di come tutto questo venga combinato per realizzare i giudizi riguardanti un tipo, è possibile osservare una definizione completa per Σ nei frammenti di codice 2.1 e 2.2.

```
1  ofType (setSigma B C) KIND IE :-
2      ofType B KIND1 IE,
3      pi b\ locDecl b B => ofType (C b) KIND2 IE,
4      pts_fun KIND1 KIND2 KIND.
5
6  of (pair B C BB CC) (setSigma B C) IE :-
7      ofType B _ IE,
8      ofType (C BB) _ IE,
9      isa BB B IE,
10     isa CC (C BB) IE.
11
12 of (p1 Pair) B IE :- isa Pair (setSigma B _) IE.
13 of (p2 Pair) (C (p1 Pair)) IE :- isa Pair (setSigma _ C) IE.
14
15 of (elim_setSigma Pair MTy M) (MTy Pair) IE  :-
16     isa Pair (setSigma B C) IE,
17     pi z\ locDecl z (setSigma B C) => ofType (MTy z) _ IE,
18     pi x y\ (locDecl x B, locDecl y (C x)) =>
19         isa (M x y) (MTy (pair B C x y)) IE.
```

Frammento 2.1: giudizi di tipaggio

```
hstep (p1 Pair) Bb :- hnf Pair (pair _ _ Bb _).
hstep (p2 Pair) Cc :- hnf Pair (pair _ _ _ Cc).
hstep (elim_setSigma Pair _M Mm) (Mm Bb Cc) :-
    hnf Pair (pair _B _C Bb Cc).

dconv (setSigma B C) (setSigma B' C') :-
    conv B B', pi x\ locDecl x B => conv (C x) (C' x).

dconv (pair B C BB CC) (pair B' C' BB' CC') :- !,
    conv B B',
    pi x\ locDecl x B => conv (C x) (C' x),
    conv BB BB',
    conv CC CC'.

dconv (elim_setSigma P MTy M) (elim_setSigma P' MTy' M') :-
    conv P P',
    isa P (setSigma B C) _,
    pi z\ locDecl z (setSigma B C) => conv (MTy z) (MTy' z),
    pi x y\ (locDecl x B, locDecl y (C x)) =>
        conv (M x y) (M' x y).

% Eta rule
dconv X (pair B C BB CC) :-
    isa X (setSigma B C) IE_, conv (p1 X) BB), conv (p2 X) CC.
```

Frammento 2.2: giudizi di conversione e riduzione

2.2 Un *interactive theorem prover* in ELPI

Estendere quanto esposto per realizzare un dimostratore interattivo ha significato principalmente due cose: implementare un elaboratore per *Minimalist Type Theory* – riutilizzando, quanto più possibile, il lavoro già fatto – e creare una procedura di dimostrazione che sia indipendente dalla teoria sottostante, ma fornisca anche degli *hooks* in cui essa possa inserirsi per usufruire delle funzionalità.

Il progetto[1], quindi, si struttura principalmente su due *directories* che contengono il codice rispettivamente di elaborazione e del dimostratore interattivo, ossia `refinement` e `itp_components`. Per la dimostrazione è presente anche un ulteriore file, `itp.elpi`, che definisce alcuni predicati interfaccia per avviare il *main loop*. Inoltre, è stato necessario modificare codice pregresso, vale a dire alcune definizioni base presenti in `main.elpi` e nei file contenenti le implementazioni dei singoli tipi in `calc`.

Il primo passo è consistito nell'estensione dei giudizi di *type inference* e *type checking* a quelli mostrati nella sottosezione 1.2.2. In particolare, l'idea fondamentale è quella di riuscire a non re-implementare interamente le regole per l'elaborazione ma semplicemente, utilizzando l'approccio di Guidi et al. (2018), dichiarare dei predicati che da un lato definiscano il raffinamento per ogni termine e dall'altro estendano il tipaggio anche alle metavariabili. L'approccio potrebbe sembrare rischioso, poiché fonde il codice di *kernel* ed elaboratore, apparentemente violando il criterio di deBruijn. Tuttavia, come hanno mo-

[1]Che può essere trovato su GitHub a questo *link*: `https://github.com/BombayAGoGo/Minimalist-Type-Theory-In-Lambda-Prolog/tree/itp`

strato Guidi et al. (2018), in presenza di termini chiusi il codice riguardante l'elaborazione diventa codice morto.

I predicati `of` e `isa` diventano perciò quaternari, includendo un ulteriore output che corrisponde al termine raffinato. Per di più, precedentemente all'estensione, `of`, `ofType`, `isa` e `isaType` non gestivano il caso metavariabile, lanciando semplicemente un errore. Questo comportamento è stato modificato permettendo la corretta sospensione dei *constraint*.

Successivamente, si è reso necessario estendere tutti i giudizi di tipaggio presenti. Da un lato, com'era prevedibile, è stato necessario modificare i predicati di tipaggio per gestire il raffinamento, dall'altro, poiché a questo punto il codice ha a che fare con metavariabili, gran parte di quanto scritto è diventato codice *unsafe*, ossia potenzialmente labile di uscita dal frammento L_λ. Un esempio di questo codice, già riportato in precedenza, è il seguente:

```
1 of (elim_setSigma Pair MTy M) (MTy Pair) IE  :-
2     isa Pair (setSigma B C) IE,
3     pi z\ locDecl z (setSigma B C) => ofType (MTy z) _ IE,
4     pi x y\ (locDecl x B, locDecl y (C x)) =>
5         isa (M x y) (MTy (pair B C x y)) IE.
```

Nella prima riga, il tipo di ritorno di El_Σ, ovvero `MTy`, essendo un tipo dipendente è, giustamente, applicato a `Pair` per ritornare il tipo effettivo, che dipende per l'appunto dalla coppia eliminata. Fintanto che `MTy` è un termine chiuso, λProlog effettuerà la β-riduzione in maniera automatica. Eppure, non appena `MTy` è una metavariabile, ad esempio se il tipo non è stato specificato dall'utente, quell'applicazione diventa `(MTy (pair B C X Y))`, che è chiaramente fuori dal frammento.

Questo problema non è limitato solamente ai giudizi di tipaggio, ma anche

a quelli di riduzione e conversione, tutti basati sull'assunzione di non avere mai a che fare con termini aperti. Ad ogni modo, va evidenziata la possibilità di ovviare al problema in questa circostanza, come effettivamente è stato fatto, assumendo di non chiedere mai la riduzione di termini aperti, che viene invece gestita separatamente dall'unificazione.

Il codice mostrato in precedenza subisce queste trasformazioni:

```
1  of (elim_setSigma P MTy M) MTy_P (elim_setSigma P' MTy M') IE :-
2      isa P (setSigma B C) P' IE,
3      pi z\ locDecl z (setSigma B C) => ofType (MTy z) _ IE,
4      pi x y\ ((locDecl x B, locDecl y (C x)) =>
5          isa (M x y) MTy_locPair (M' x y) IE,
6          pi p\ (locDecl p (setSigma B C), copy p (pair B C x y))
7              => conv (MTy p) MTy_locPair
8      ),
9      pi p\ (locDecl p (setSigma B C), copy p P')
10         => conv (MTy p) MTy_P.
```

Frammento 2.3: `of` con *refinement*

Innanzitutto, è osservabile come sia `of` che `isa` ritornino termini elaborati – `P'`, `M'` e `elim_setSigma P' MTy M'`. Poi, si può notare l'approccio utilizzato per evitare la fuoriuscita dal frammento, ispirato al metodo utilizzato da Miller (1991b) con i predicati `copy/subst`. Infatti, ogni qualvolta si vorrebbe applicare una meta-λ-astrazione– i.e. `x\...` – a un termine, questo viene sostituito da una variabile fresca e vengono aggiunte due ipotesi: la prima dichiara la variabile *placeholder* come essere dello stesso tipo del termine sostituito, la seconda stabilisce una relazione di copia tra *placeholder* e termine. La dichiarazione di tipaggio è necessaria per la *type inference* del sottotermine mentre quella di copia viene utilizzata, eventualmente, durante l'unificazione.

Nonostante questa rielaborazione sia più verbosa, le va riconosciuto il merito di funzionare correttamente sia in presenza di metavariabili, quindi durante l'elaborazione, sia per i termini chiusi, ottenendo così di poter ovviare al problema di duplicazione di codice tra *kernel* e `refiner`.

2.2.1 *Refinement*

Per quanto riguarda l'elaborazione dei termini base, come si è visto nel frammento 2.3, ogni regola non fa altro che ritornare i risultati dei sottotermini e, dal momento che variabili locali e costanti si raffinano in sè stesse (`of X T X _ :- locDecl X T.`), il comportamento che si ottiene finora è quello di lasciare intoccati i termini.

Il vero uso di questa *feature* viene effettuato da termini che estendono quelli normali, ad esempio con informazioni aggiuntive come i nomi delle variabili dei *binders* – utili durante la dimostrazione assistita per permettere all'utente di potervisi riferire – oppure addirittura con termini completamente nuovi, come il costrutto `app X dots`, introdotto da Asperti, Ricciotti et al. (2012).

```
1 isa (app Lam dots) T O IE :- isa Lam T O IE.
2 isa (app Lam dots) T O IE :- !,
3     isa Lam (setPi _ _) _ _,
4     isa (app (app Lam Arg) dots) T O IE.
```

`app X dots` corrisponde all'applicazione di una variabile a zero o più argomenti, determinati dal tipo di `X`: se è un tipo semplice, allora l'applicazione di `X`

è in realtà l'uso di `X` stessa; se è una funzione – o un abitante del tipo Π in *Minimalist Type Theory* – allora vengono generate tante metavariabli quanti sono gli argomenti necessari affinché l'applicazione abbia il tipo atteso. Un termine contente `dots` non fa parte, chiaramente, di quelli definiti dalla teoria; tuttavia tramite il processo di raffinamento è possibile estendere il linguaggio con qualsiasi costrutto possa facilitare il lavoro dell'utente.

Un altro uso della *feature* di raffinamento è dato dalle *coercion*. Il seguente è un esempio ispirato all'approccio di Guidi et al. (2018).

```
1  coerce bool nat (x\ ite x (succ zero) zero).
2  coerce A B F :-
3      coerce A Mid F', coerce Mid B F'',
4      F = (x\ app F'' (app F' x)).
5
6  isa Term Type Out IE :-
7      of Term Type' _,
8      coerce Type' Type F,
9      subst F [Term] Coerced,
10     isa Coerced Type Out.
```

Il *type checking* è esteso con una clausola che considera la possibilità di trasformare un termine tramite delle *coercion* definite dall'utente, che possono essere assunte a *runtime* tramite l'implicazione oppure definite in un qualche file apposito. Ogniqualvolta si chiede di effettuare il *type checking* scatterà la clausola base, che verificherà la convertibilità del tipo inferito e di quello atteso. Qualora questa fallisse, si considererebbe la possibilità dell'esistenza di una *coercion*, o meglio di una composizione di *coercion*. L'uso di un termine in presenza di *mismatch* di tipo, quindi, verrebbe raffinato automaticamente a un termine ben tipato.

Unificazione

Come già trattato in precedenza, parte del compito del *refiner* consiste nell'effettuare unificazione dei termini, che può rendersi necessaria durante la *type inference* – poiché si è in presenza di tipi dipendenti – oppure può essere richiesta esplicitamente dall'utente.

```
1 macro @func A B :- setPi A (_\ B).
2
3 pi u v\ (locDecl u (@func nat nat), locDecl v nat)
4     => conv (app X (app u v)) (app u (app X v)).
```

Frammento 2.4: $\forall u, v. X\ (u\ v) \approx u\ (X\ v)$

In questo esempio, sotto opportune ipotesi di tipaggio – `u` è una funzione non dipendente da $\mathbb{N}$ e $\mathbb{N}$ e `v` $\in \mathbb{N}$ – si sta chiedendo di convertire i due termini. Se si seguissero le regole di conversione, ciò porterebbe a tentare di convertire le sottoparti del termine, per la regola

```
dconv (app M N) (app P Q) :- conv M P, conv N Q.
```

Questo si tradurrebbe in `conv X u` e `conv (app u v) (app X v)`, le quali porterebbero `X` a unificare con `u`. Sebbene questo comportamento sia corretto, almeno in linea di principio, non corrisponde esattamente a quanto atteso. Questo problema di unificazione infatti ha infinite soluzioni della forma:

```
1 lambda nat (x\ x)
2 lambda nat (x\ app u x)
3 lambda nat (x\ app (app u x)) ...
```

L'implementazione di una sorta di unificazione del "prim'ordine" modulo α-conversione va estesa con un algoritmo che sia in grado risolvere una classe di problemi più ampia possibile, nel modo più generale possibile, tenendo sempre presente che se ne sta affrontando uno indecidibile.

Per fare questo, ogni dimostratore implementa un algoritmo che incorpora un certo numero di euristiche – si veda Ziliani e Sozeau (2015) per Coq e de Moura et al. (2015) per Lean – che, per quanto possano variare anche sensibilmente, seguono ugualmente alcune idee comuni, ad esempio quelle tratte dal lavoro di Huet (1975). A questo proposito occorre sottolineare che vi sono quattro classificazioni possibili per un problema di unificazione: *rigid-rigid*, *rigid-flexible*, *flexible-rigid* e *flexible-flexible*. Questa classificazione dipende dalla testa dei termini coinvolti nel problema, ossia il primo termine che si incontra in un'applicazione o nel corpo di una λ-astrazione. Se il termine di testa è una metavariabile si ha un termine a testa flessibile; negli altri casi è a testa rigida. Si prenda l'esempio precedente:

```
1 app X (app u v).
2 app u (app X v).
```

Entrambi i termini sono applicazioni, ma il primo – $X\ (u\ v)$ – ha come elemento di testa `X`, una metavariabile, mentre il secondo – $u\ (X\ v)$ – ha `u`, che è un termine rigido.

Gli unici casi che richiedono di essere gestiti fuori dalla conversione sono quelli in cui almeno uno dei due termini è a testa flessibile. I casi *flexible-rigid* e *rigid-flexible* sono simmetrici e, tendenzialmente, se ne considera uno mentre l'altro viene semplicemente portato nell'altra forma. I casi *flexible-flexible* presentano invece numerose problematiche, poiché corrispondono a un

tentativo di unificazione di due metavariabili, un problema estremamente complesso e che ha potenzialmente un'infinità di soluzioni. L'approccio adottato tradizionalmente consiste nel sospendere per poi tornare a verificare qualora le metavariabili coinvolte venissero istanziate.

Per quanto riguarda i casi *rigid-rigid*, ogni algoritmo presenta una sorta di unificazione del "prim'ordine" che decompone i termini della stessa forma in sottotermini, andandone a verificare l'unificabilità ricorsivamente. In altre parole, si considerano i casi applicazione/applicazione, λ/λ eccetera e li si decompone. Questo è già, di fatto, ciò che fa la conversione.

Il primo termine dell'esempio 2.4, secondo la terminologia di de Moura et al. (2015), è *stuck*. Se viene richiesta la `conv` fra termini di cui almeno uno è a testa flessibile, è necessario procedere con l'unificazione.

```
1 conv X Y :- is_stuck X, is_stuck Y, !,
2     flex_flex_constraint X Y X Y.
3 conv X Y :- (is_stuck X, !; is_stuck Y), !,
4     unwind_app X XApp XArgs,
5     unwind_app Y YApp YArgs,
6 unify XApp XArgs YApp YArgs.
```

Questo è lo *snippet* che estende il predicato di conversione aggiungendo una clausola che controlla se si è in presenza di conversione *flexible-flexible* o *flexible-rigid* – e viceversa. Nel primo caso la conversione viene sospesa per poi riprendere qualora i termini di testa venissero istanziati; nel secondo caso i termini si trasformano per separare la testa dalla lista di argomenti tramite `unwind_app`. Ad esempio un termine come `app (app (app x) y) z)` viene separato in `x` e `[y, z]`. Tutto ciò serve, in particolare, per istanziare metavariabili con argo-

menti in λ-astrazioni e aumentare il programma con predicati `copy` che possono essere usati durante fase di *projection*.

```
1 unify (uvar as X) [L|Ls] T L' :- !,
2     of L LTy _ _,
3     X = lambda LTy F,
4     pi x\ (locDecl x LTy, copy x L)
5         => unify (F x) Ls T L'.
```

Ogni algoritmo di unificazione implementa almeno due euristiche, descritte per la prima volta da Huet (1975): *mimic* e *projection*. La prima tenta di risolvere un problema *flexible-rigid* imitando il termine rigido nel termine flessibile, vale a dire istanziando il termine flessibile con un termine della stessa forma per poi ricorsivamente unificare i sottotermini. La seconda euristica, al contrario, tenta di risolvere l'unificazione proiettando uno degli argomenti che sono in input alla metavariabile. Un esempio di queste euristiche è stato già esposto parlando della risoluzione di problemi nel *pattern fragment*: in quel contesto non si parlava di euristiche poiché se si è in L_λ non vi sono scelte, la soluzione avviene per imitazione oppure per proiezione oppure non esiste. In un contesto più generale, invece, multiple. La maggior parte delle euristiche tende a favorire la proiezione alla mimica.

La regola mostrata nel frammento 2.2.1 scorre tutti gli argomenti forniti in input alla metavariabile che rendeva il termine *stuck*, istanziandola ogni volta con una λ-astrazione per poi andare in ricorsione sulla metavariabile che rappresenta il corpo. Una volta esauriti gli argomenti, si ottiene una metavariabile applicata a una variabile fresca per ciascun argomento e si considera se sia possibile proiettare su una delle variabili locali a cui è applicata.

```
1 mode (try_projection i i i).
2 try_projection _ [] _ :- fail.
3 try_projection X [V|Vs] T :-
4     (not (forbid V),
5     get_copy V T',
6     use_var V T' T Asgn,
7     X = Asgn);
8     try_projection X Vs App.
```

Proiettare in assenza di argomenti porta a un fallimento, altrimenti tramite `get_copy` si ottiene, in `T'`, il termine a cui la variabile locale si copia, che viene poi unificato con `T` utilizzando `use_var` il quale, a sua volta, produce un assegnamento per `X`. Se ciò ha successo allora la clausola si risolverà, altrimenti tenterà una proiezione su un altro argomento.

Come accennato, `use_var` implementa ulteriori euristiche che considerano l'unificazione attraverso l'uso del termine preso in input. Ad esempio si consideri `conv (app X (pair a (_\b) x y)) x`. Senza ulteriori euristiche, il problema verrebbe risolto per mezzo della mimica di `x` – posto che sia una variabile globale, altrimenti non vi sarebbe nemmeno soluzione. Si può osservare, tuttavia, che una soluzione piuttosto sensata sarebbe `X = lambda (setSigma a (_\b)) (p\p1 p)`, ossia effettuare la prima proiezione sulla coppia. `use_var`, data una variabile locale, il termine a cui si copia, e il termine a cui si deve unificare, attraverso il *pattern matching* sul termine copiato tenta di costruire un termine che usi quella variabile.

```
1 use_var V (pair _ _ P _) S O :- use_var (p1 V) P S O.
2 use_var V (pair _ _ _ Q) S O :- use_var (p2 V) Q S O.
3
4 use_var V T S V :- conv T S.
```

L'esempio consente di osservare le euristiche per l'uso di un termine coppia. Le prime due tentano di costruire proiezioni mentre la terza rappresenta il caso base.

Infine, se la proiezione non ha successo, viene testata la mimica, sia per i termini che per i tipi.

Un ultimo aspetto che riguarda l'unificazione è il controllo della riduzione. È stato detto che l'*entry point* dell'unificazione è il controllo per i termini *stuck* e cioè a testa flessibile. Va notato che in questa categoria è importante considerare anche termini come `conv (app (lambda nat F) (app f x)) (app f x)`. Ciò perché se non si bloccasse la conversione, questa tenterebbe di normalizzare il primo termine applicando `F` a `app f x` e producendo un problema esterno al *pattern fragment*. Per evitarlo, sono stati aggiunti diversi predicati il cui scopo è di catturare i redessi dove il *binder* è una termine non chiuso, effettuando una sorta di riduzione controllata.

```
1  macro @unify T S :- unify T [] S [].
2
3  conv (app (lambda A F as Lam) X) T :-
4      (pi x\ not (is_closed (F x))), !,
5      @unify (app Lam X) T.
6
7  unify (app (lambda A F) X) L T L' :- !,
8      pi x\ (locDecl x A, copy x X) => unify (F x) L T L'.
9
10 unify (app Lam X) L T L' :-
11     name Lam, get_copy Lam Lam', Lam' = lambda A F,
12     pi x\ (locDecl x A, copy x X, forbid Lam)
13         => unify (F x) L T L'.
```

Il primo predicato, `conv`, lancia l'unificazione in presenza di un β-redesso; gli altri due, quelli di `unify`, gestiscono la riduzione vera e propria: il primo nel caso in cui effettivamente sia presente un β-redesso mentre il secondo nel caso in cui sia stato inserito un *placeholder* al posto di una λ-astrazione durante la ricorsione su un *binder*.

Per concludere va sottolineato come, grazie all'uso di λProlog, non sia stato necessario implementare la gestione di unificazione nel caso di problemi in L_λ, che oltre a essere già parte del linguaggio diventa la forza trainante dell'algoritmo euristico.

2.2.2 Dimostrazione interattiva

La componente di dimostrazione interattiva si articola in due parti, una che implementa le *routine* di gestione del processo interattivo vero e proprio mentre l'altra che è costituita, come anticipato, da una serie di *hooks* implementati appositamente per *Minimalist Type Theory* e che specificano funzionalità dipendenti dal linguaggio come costruzione di *goal* a partire da termini e tattiche primitive.

Prima di procedere alla descrizione dei predicati implementati, verrà mostrato un esempio di dimostrazione in modo da poterli contestualizzare. Si supponga di voler dimostrare $(\Pi x \in A)(B(x) \to A)$, corrispondente a $A \supset B \supset A$. Questo tipo, tradotto in λProlog, è (`setPi A (x\setPi (B x) (_\A))`).

Prima di poter lanciare il *main loop*, è necessario chiamare il giudizio di *type inference* `of X (x\setPi (B x) (_\A)) Y IE`. Questo porterà alla creazione di un *constraint*

```
[] ?- isa X (x\ setPi (B x) (_\ A)) Y IE.
```

La procedura vera e propria può essere poi lanciata chiamando il *main loop* con `itp_loop`

```
> itp_loop X (state
    [(1, goal X (x\ setPi (B x) (_\ A)) Y [])] [] [] 1)
    [] OutGoals OutScript.
```

Come si può osservare, `itp_loop` opera su uno stato contenente una lista di *goal* numerati, più ulteriori strutture per i *tinycal*. Come input può ricevere script sotto forma di lista di comandi – vuota nell'esempio – e restituisce *goal* e lo script totale in output. Ogni *goal* mantiene la metavariabile da istanziare – `X` – il tipo, la metavariabile a cui questa si raffina – `Y` – e una lista di dichiarazioni di variabili, utilizzata per memorizzare il contesto.

Poiché non vi è nessun comando in input, la procedura chiederà all'utente di immetterne uno. Dovendo creare un abitante del tipo Π, l'unica possibilità è quella di utilizzare la regola di introduzione

```
"you got 1 open goal"
"1) [] |- X :: setPi(A, x\ setPi(B x, _\ A))"
"input command"
> intro x
```

che produce

```
itp_loop X (state
    [(2, goal F (setPi (B x) (_\ A)) F' [(vdecl x A)])] [] [] 2)
    [] OutGoals [intro x].
```

La regola di introduzione porta alla creazione di un nuovo *goal* in seguito all'istanziazione `X = lambda A F`, che introduce una variabile `x` di tipo `A` nel contesto del sequente e apre `F` come *goal* corrente. L'istanziazione inoltre avrà risvegliato il tipaggio, producendo un *constraint* della forma

```
[locDecl c0 A] ?- isa (F c0) (setPi (B c0) (_\ A)) (F' c0) IE.
```

dove `c0` è una variabile fresca.

Di nuovo, poiché si è in presenza del tipo `setPi`, l'unica possibilità è utilizzare nuovamente l'introduzione. Se non si è interessati a dare un nome alle variabili, ad esempio poiché non verranno utilizzate, è possibile utilizzare la versione dei comandi senza introduzione dei nomi, come `intro`. Il comando produrrà comunque una variabile del tipo `B`, tuttavia questa avrà un nome assegnato automaticamente.

```
"you got 1 open goal"
"2) [(x: A)] |- F [x]: setPi(B x, _\ A)"
"input command"
> intro

itp_loop X (state
    [(3, goal G A G' [(vdecl x A), (vdecl lambda_0 (B x))])]
    [] [] 3) [] OutGoals [intro x, intro].
```

E corrispondentemente il tipaggio avrà prodotto

```
[locDecl c0 A, locDecl c1 (B c0)] ?-
    isa (G c0 c1) (_\ A) (G' c0 c1) IE.
```

Infine, poiché bisogna fornire un abitante del tipo `A`, e dato che `x` ha proprio tipo `A`, tramite la regola di applicazione è possibile usare la variabile `x` per concludere la dimostrazione.

```
"you got 1 open goal"
"3) [(x: A), (lambda_0: B x)] |- G [x, lambda_0]: A"
"input command"
> app x

itp_loop X (state
    [] [] [] 4)
    [] [] [intro x, intro, app x].
```

L'uso di `x` corrisponderà all'assegnamento `G c0 c1 = c0`. Il termine raffinato totale sarà `Y = lambda A (x\lambda (B x) (_\x))`

L'*entry point* del dimostratore, come mostrato, è `itp_loop`, che permette di avviare la procedura dopo aver lanciato la *type inference* su una metavariabile, dato un tipo concreto che rappresenta la formula da dimostrare. Infatti, chiamando `of X T PT IE` con `X` metavariabile, si causa la creazione di un *constraint* che asserisce che `X` deve essere di tipo `T`. Questo fa si che, partendo da `X`, il dimostratore non si debba preoccupare del tipaggio, che avviene in *background*, dovendo semplicemente consentire l'istanziazione delle metavariabili che rappresentano i *goals*, tenendo traccia di esse.

Gli argomenti di `itp_loop` sono, più precisamente: una metavariabile, che alla fine della procedura conterrà il termine (raffinato) del tipo corretto; uno stato, composto da lista di *goals* aperti e altre informazioni usate per l'implementazione dei *tinycals*; uno script sotto forma di lista di *tinycal*, tattiche e tatticali, che vengono applicati in sequenza. Questo predicato produce inoltre due output: i *goals* rimanenti, qualora la procedura venisse interrotta prima di averli chiusi tutti e lo *script* totale, fatto di un eventuale *script* in input e dei comandi usati dall'utente per completare la prova una volta esauriti quelli dati in input.

```
1  itp_loop _ State _ [] [] :- empty_state State.
2
3  itp_loop Term State [] OutGoals OutScript :-
4      print_status State,
5      prompt_user "input tactic" Tac,
6      (Tac = undo, !, fail;
7      Tac = quit, get_outgoals Term State OutGoals, !;
8      process_tac Tac State NewState,
9      append [Tac] OutScript' OutScript,
10     itp_loop Term NewState [] OutGoals OutScript';
11     print "error with" Tac,
12     itp_loop Term State [] OutGoals OutScript).
13
14 itp_loop Term State [Tac|Tacs] OutGoals OutScript :-
15     process_tac Tac State NewState,
16     append [Tac] OutScript' OutScript,
17     itp_loop Term NewState Tacs OutGoals OutScript'.
```

Il caso base è quello in cui lo stato è vuoto e non vi sono ulteriori tattiche da eseguire in input. Se lo stato non contenesse *goals* da dimostrare ma vi fossero comunque tattiche rimanenti questo risulterebbe in un errore.

Nel caso in cui vi siano *goals* da dimostrare ma nessuno *script* in input,

dopo aver stampato una rappresentazione dello stato corrente all'utente, viene chiesto di dare un comando. Il comando `undo` causa il fallimento del predicato che porta a fare *backtracking* allo stato precedente, annullando l'ultimo input. `quit` invece interrompe una dimostrazione prima del completamento e rende necessario raccogliere i *goals* rimasti. Quest'ultima operazione non è triviale come potrebbe apparire, poiché alcuni *goals* potrebbero essere diventati non più raggiungibili (ad esempio se una riduzione ha cancellato una metavariabile da un termine). Ciò porta alla necessità di effettuare un controllo di raggiungibilità a partire da `Term`, per verificare quali metavariabili vi compaiano rispetto a quelle rimaste nello stato. Questo controllo dipende dai termini della teoria, che deve implementare l'*hook* `meta_in_term`.

Se l'input non è né `undo` né `quit`, esso è passato a `process_tac` che lo interpreta. Questo predicato riceve come input un comando e lo stato corrente e ritorna lo stato modificato. Tramite `process_tac` sono implementati *tinycal*, tatticali LCF e, qualora il comando inserito non corrispondesse a nessuno di questi, il caso base che mappa una tattica primitiva a ogni *goal* presente nello stato.

```
1  process_tac Tac (state Goals I L M) (state NewGoals I L M') :-
2      unlabel_goals Goals UGoals,
3      bind_ltac Tac UGoals NewUnlabeledGoals,
4      label_goals M NewUnlabeledGoals M' NewGoals.
5
6  mode (bind_ltac i i o).
7  bind_ltac _ [] [].
8  bind_ltac Tac [Goal|Goals] NewGoals :-
9      exec_tac Tac Goal NewGoals',
10     append NewGoals' NewGoals'' NewGoals,
11     bind_ltac Tac Goals NewGoals''.
```

I *goal* sono rappresentati come `goal X T Y D` dove `X` è la metavariabile da istanziare, `T` è una rappresentazione del tipo, usata principalmente per dare informazioni all'utente, `Y` è una metavariabile che funge da puntatore al termine raffinato – ottenuto istanziando `X` – e `D` è una lista di variabili locali, che contengono nome della variabile e rappresentazione del tipo. Di fatto, sono una rappresentazione del sequente associato alla metavariabile `X`, che si può immaginare come $D \vdash X : T \rightsquigarrow Y$.

Va chiarito cosa si intende per rappresentazione del tipo e come avviene in effetti l'istanziazione delle metavariabili. Su ogni metavariabile `X` presente in un *goal* è definito un *constraint* di tipaggio più o meno della forma:

```
locDecl c0 tc0, ..., locDecl cn tcn ?-
    of (X c0 ... cn) T (Y c0 ... cn) _.
```

Vale a dire che è presente un *constraint* canonico in cui la metavariabile compare applicata a delle variabili locali. La difficoltà sta nel fatto che quelle variabili locali sono in un contesto differente rispetto alla procedura di dimostrazione, poiché sono generate dal tipaggio la prima volta che viene effettuata la *type inference* della metavariabile. Per esempio, si supponga di chiamare `of X (setPi nat (_\nat)) Y _`. Questo lancia la procedura di tipaggio che, dal momento che `X` è una metavariabile, si bloccherà con la dichiarazione di un *constraint*

```
[] ?- isa X (setPi nat (_\ nat)) Y _.
```

Poiché `X` deve avere tipo `setPi`, ossia quello delle funzioni dipendenti, l'unica

possibilità è quella di istanziarla tramite un assegnamento a `lambda T F`, che risveglia il *constraint* mostrato e arriva a porne un altro

```
[locDecl x nat] ?- isa (F x) nat (G x) _.
```

A questo punto, come si può vedere, la metavaribile correntemente sotto esame, `F`, si trova in un *constraint* che ha senso solo in presenza di una variabile locale `x` a cui essa è applicata e che non è accessibile dal dimostratore. Questo porta ad avere due alternative dal punto di vista dell'*interactive theorem prover*. La prima è tenere la metavariabile aggiornata anche dal lato del dimostratore, ossia ogniqualvolta una tattica assegna un termine che crea variabili locali – come `lambda` – si procede in ricorsione su *goal* contenenti metavariabili applicate a queste. Un ipotetico codice potrebbe essere

```
1 tactic (intro N) (goal X (setPi S T) Y Vars) :-
2     X = lambda S F, Y = lambda S G,
3     pi x\ loop (goal (F x) (T x) (G x)) [(x, N)|Vars].
```

La seconda alternativa è mantenere sempre la metavariabile memorizzata nel *goal* non applicata, tenendo però traccia di quante variabili fresche che è necessario creare per allinearla col *constraint*.

La prima soluzione, per quanto più naturale, obbligherebbe di fatto l'utente a dimostrare i *goals* in modo *depth-first*, senza poter passare liberamente da uno all'altro.

La seconda soluzione è, invece, per certi versi più complessa, poiché richiede di ricreare il contesto per ciascuna metavariabile ogni volta che vi si vuole assegnare un termine tramite una tattica. Tuttavia, è anche quella che offre

più flessibilita all'utente. Per questo è la soluzione scelta.

```
1  mode (exec_tac i i o).
2  exec_tac Tac (goal X T Y V as Goal) NewGoals :-
3      exec_tac_aux Tac Goal V [] NewGoals.
4
5  mode (exec_tac_aux i i i i o).
6  exec_tac_aux Tac (goal X T Y D as G) [] FVars NG :-
7      tactic Tac G FVars Term, X = Term, build_newgoals Y D NG.
8
9  exec_tac_aux Tac (goal X T Y D) [vdecl N TN|Ds] FVs NG :-
10   pi x\ copy_var x N => (
11     copy_to_type TN TN',
12     locDecl x TN' =>
13       exec_tac_aux Tac (goal (X x) T (Y x) D) Ds [(x, N)|FVs] NG
14     ).
```

`exec_tac`, utilizzato da `bind_ltac`, è il predicato responsabile della riproduzione del contesto di un goal per poi eseguire effettivamente una tattica. Nello specifico, scorre la lista di tutte le dichiarazioni di variabili, che rappresentano le corrispettive dal lato del dimostratore, e per ognuna dichiara una variabile fresca a cui applica la metavariabile. Una volta terminate le dichiarazioni, viene chiamata la tattica primitiva, che ritornerà il termine con cui istanziare `X`. Successivamente, utilizzando l'altra parte del sequente, cioè la metavariabile a cui `X` si raffina, `Y`, si costruiscono i nuovi *goals* tramite l'*hook* `build_newgoals`.

Il motivo per cui si recuperano i *goals* a partire da `Y` e non da `X` è da ricercarsi nell'istanziazione, poiché se avenisse con un termine `dots` allora le metavariabili generate durante il raffinamento andrebbero perse, poiché non sono contenute in alcun modo in `X`. Viceversa, `Y` può essere visto come un puntatore al termine raffinato, qualunque esso sia. In questo modo, quando l'assegnamento a `X` è terminato, `Y` conterrà sempre un termine completo in cui

sono presenti tutte le metavariabili create.

La costruzione dei nuovi *goals*, essendo dipendente dalla teoria, dev'essere implementata lato linguaggio.

```
1  mode (build_newgoals i i o).
2  build_newgoals (uvar K L as Y) VDecls NewGoals :- !,
3      shave_vars L VDecls L',
4      lookup Y VDecls X T,
5      apply_meta K L' Y',
6      NewGoals = [goal X T Y' VDecls].
7
8  build_newgoals (lambda T F) VDecls NewGoals :-
9      copy_from_type T T',
10     get_fresh_name lambdav N VDecls,
11     append VDecls [vdecl N T'] L,
12     pi x\ build_newgoals (F x) L NewGoals.
13
14 build_newgoals (lambda N T F) VDecls NewGoals :-
15     not_a_dup N VDecls,
16     copy_from_type T T',
17     append VDecls [vdecl N T'] L,
18     pi x\ build_newgoals (F x) L NewGoals.
19
20 build_newgoals (app M N) VDecls NewGoals :-
21     build_newgoals M VDecls NewGoals',
22     build_newgoals N VDecls NewGoals'',
23     append NewGoals' NewGoals'' NewGoals.
```

`build_newgoals` non fa altro che andare in ricorsione sui termini per creare i nuovi sequenti. Per i termini che introducono variabili, come `lambda`, questo implica la creazione di una dichiarazione a cui viene associato un nome – dato in input dall'utente, caso `lambda` con nome, o generato automaticamente – e un tipo, o meglio la traduzione del tipo ottenuta sostituendo tutte le variabili

fresche create durante la `exec_tac` con i nomi corrispondenti – ottenuti dalla clausola `copy_var x N`, inserita durante la `exec_tac`. Per i termini che non introducono nessuna variabile il codice ammonta a delle semplici chiamate ricorsive.

Il predicato base è quello in cui si considera una metavariabile vera e propria. In tal caso questa, poiché ottenuta da un termine raffinato, sarà a sua volta il raffinato di un'altra metavariabile, che vorremmo descrivere con un sequente attraverso un `lookup` che tramite una regola CHR accede al contesto dei *constraint* di tipaggio per recuperare `X`.

```
1  constraint isa read-refined of {
2    rule (E1 ?- isa (uvar K L1) Ty (uvar O L1' as Y) _)
3    \ (E2 ?- read-refined (uvar O L2) VDecls X T)
4    | (L1 = L1',
5      reverse L1' RL1', reverse L2 RL2, reverse VDecls RVDecls,
6      create_copy_hyps RVDecls RL2 RL1' Hyps Rem,
7      Hyps => copy_from_type Ty S, !;
8      print "align fail on" L1 "=" L1', halt)
9    <=> (E2: (X = uvar K Rem, T = S)).
10 }
```

Questa regola viene risvegliata attraverso la dichiarazione di un *constraint* su `read-refined Y D X T`. Il suo scopo principale, dettagli implementativi a parte, è di fare *match* tra la `Y` di `read-refined` e la stessa `Y` di un *constraint* `isa X Ty Y _` e in questo modo ottenere sia `X` che il suo tipo.

Infine, le tattiche primitive, vale a dire quelle che corrispondono alle regole di introduzione ed eliminazione dei connettivi, sono definite con la *signature* `tactic Name Goal FVars OutTerm`, di cui i primi tre sono input e l'ultimo

un output. `Goal` permette l'accesso al tipo della metavariabile, che può essere utilizzato da certe tattiche per controllare di star istanziando una metavariabile del tipo corretto. `FVars` contiene le variabili locali introdotte con `exec_tac` e permette alle tattiche di usare una variabile definita nel contesto.

Alcune tattiche creano termini che necessitano di nomi, ma non sempre l'utente è interessato a specificarne uno: si pensi a $\forall A.A \Rightarrow A$, dove chiaramente il $\forall$ crea un *binding* per una variabile di tipo A. Come scelta implementativa si è deciso allora di imporre che tutte le tattiche ritornino termini con nome, se questi necessitano di uno. Come `build_goals`, i nomi sono generati automaticamente.

```
1 tactic intro (goal _ (setPi _ _) _ V) _ (lambda N A F) :-
2     get_fresh_name lambdav V N.
3
4 tactic (intro N) (goal _ (setPi _ _) _ _) _ (lambda N A F).
5
6 tactic (app N) _ FVars (app Lam dots) :- get_named N FVars Lam.
```

Capitolo 3

Conclusione

Considerando il superamento dello stato dell'arte auspicato nell'introdurre questa trattazione, si può certamente affermare che λProlog si è rivelato estremamente promettente nel permettere di risolvere in modi migliori il *task* di creare *kernel* ed elaboratori ai fini dell'*interactive theorem proving*, sebbene vi sia ancora ricerca da fare.

Lavori futuri potrebbero riguardare la creazione di un'estesa libreria di teoremi per *Minimalist Type Theory* attraverso il dimostratore interattivo, ed eventualmente anche il *porting* di questo dimostratore per altre teorie, in modo da valutare l'effettiva modularità. Per quanto riguarda la dimostrazione interattiva, inoltre, attualmente è necessario codificare in λProlog sia tipi che termini sui quali enunciare definizioni, teoremi e dimostrazioni. Inoltre, gli script devono essere passati manualmente come liste di *tinycal*. In generale, sarebbe conveniente permettere l'uso di un qualche linguaggio intermedio in cui gli utenti possano definire script, eventualmente con una notazione più vicina a quella matematica, di cui poi effettuare il *parsing* in λProlog. In aggiunta, si

potrebbe sperimentare maggiormente con le *feature* di *refinement*, estendendolo ad esempio con costrutti come *type classes*, *canonical structures* o *unification hints*. Similmente, sarebbe interessante testare anche tattiche con un comportamento più complesso di quello delle semplici tattiche primitive, per verificare la corretta impostazione architetturale del dimostratore.

Per quanto riguarda l'algoritmo di unificazione, essendo euristico, necessiterebbe di essere testato estensivamente, se possibile anche implementando gli *hook* necessari per altri calcoli, ad esempio CIC, in modo da poter accedere a librerie di termini più estese. In secondo luogo, sarebbe proficuo aggiungere nuove euristiche, soprattutto per la gestione dei casi *flexible-flexible*. Ciò sarebbe un'ulteriore testimonianza della facilità di estensione di programmi λProlog.

Complessivamente, sarebbe opportuno migliorare la predicibilità del sistema. Creare programmi estesi in un linguaggio logico in cui ogni regola può contenere o meno *cut* – e quindi alterare la semantica della ricerca – può risvegliare *constraint*, che a loro volta riprendono l'esecuzione e possono fallire, effettuare *backtracking* e non solo, porta a un'interazione spesso caotica. Dal punto di vista dello sviluppo, tutto ciò complica la comprensione di eventuali *bug* e il loro *debugging*. Dal punto di vista dell'utente, soprattutto qualora volesse estendere il sistema, bisognerebbe garantire che eventuale codice aggiuntivo non alterasse l'esecuzione di quello già presente in modi al di là del suo controllo.

Per ciò che concerne gli aspetti posivi, sebbene vi sia spazio per ampliare il sistema, va osservato che – soprattutto in relazione alle problematiche evidenziate da Guidi et al. (2018) e all'intenzione di superare lo stato dell'arte – λProlog e, in particolare, la sua estensione ELPI, si sono mostrati più che

adeguati per risolvere gli scopi preposti. Per quanto riguarda la gestione di *binders* e metavariabili, tramite l'approccio *Higher Order Abstract Syntax* in λProlog è possibile dare una traduzione fedele delle regole di tipaggio di teorie dei tipi anche dipendenti, come nel caso di *Minimalist Type Theory*. Inoltre, effettuando l'*embedding* delle metavariabili del linguaggio oggetto in quelle del meta-linguaggio, si riesce a ottenere un supporto praticamente nativo per la realizzazione di dimostratori interattivi, soprattutto in virtù della possibilità di definire *constraint* arbitrari, offerta da ELPI. Basti pensare che la maggior parte del codice è andata verso lo sviluppo di elementi come *tinycals*, controllo dello stato e rappresentazione e manipolazione dei *goal*.

Il problema di duplicazione di codice è attenuato, poiché si è riusciti a integrare gran parte dei giudizi di *refinement* con quelli del *kernel* senza compromettere la robustezza di quest'ultimo. In aggiunta, il meccanismo di unificazione di problemi del *pattern fragment*, su cui è basato λProlog, si è rivelato estremamente utile per via della varietà di problemi più generali risolvibili attraverso di esso, ponendosi così come meccanismo centrale con cui effettuare le computazioni necessarie. Infine, il problema dell'estensibilità può considerarsi quanto meno alleviato: il linguaggio del sistema e quello a disposizione dell'utente non presentano alcun *mismatch*, come invece accade per altri sistemi. Inoltre, il linguaggio usato è di alto livello, consentendone un uso agevole. In aggiunta a questo, la struttura stessa dei linguaggi logici, fatti di clausole su cui viene effettuata una procedura di ricerca, si presta particolarmente all'estensibilità, che spesso si riduce all'inserire clausole aggiuntive nel posto giusto.

In tutto ciò, occorre sottolineare come senza le aggiunte apportate da ELPI non si sarebbe potuto portare a termine il compito. Le regole CHR hanno

assunto un ruolo chiave nella realizzazione del dimostratore, rivelandosi più che adeguate alla manipolazione dei *constraint* generati, che si è resa più volte necessaria, come nel caso del recupero delle metavariabili con `lookup`. Va sottolineato anche che queste regole hanno permesso, come "*side-effect*", di avere un controllo più fine sulle ipotesi di una clausola, ossia i predicati aggiunti tramite implicazione (`=>`). Una volta inserito un predicato in questo modo, è possibile recuperalo in maniera indiretta semplicemente scrivendo una clausola che dovrebbe fare *match* con esso, tuttavia in questo modo non si ha né controllo su quale predicato viene risvegliato per primo né sull'uso che si può fare dei predicati presenti nelle ipotesi. Grazie alle regole CHR, invece, è possibile aggregare gruppi di predicati, permettendo un'analisi più fine delle ipotesi aggiunte. Questo è stato utilizzato per l'euristica di *projection*, nella quale, spesso, bisogna risalire catene di `copy` per arrivare al termine originale.

Con `mode`, la seconda delle *feature* principali di ELPI, il codice diventa non solo più predicibile, poiché si può limitare la semantica generativa, ma anche la presenza di metavariabili, che spesso richiede codice ad hoc per essere gestita, è più controllabile.

Infine, dal punto di vista dello sforzo di sviluppo, il lavoro è stato portato a termine nell'arco di due mesi e, in termini di dimensione, ammonta a:

- circa 500 righe di codice per la parte *language independent* del *loop* di dimostrazione interattiva, compreso di *tinycals*, tatticali LCF e recupero delle metavariabili;

- poco più di 1000 righe di codice per la parte *language dependent* della dimostrazione interattiva, con costruzione di *goal*, controllo di raggiungibilità sui termini, tattiche dei connettivi e codice di copia di termini e

tipi per il passaggio dal contesto di tipaggio a quello di dimostrazione;

- 800 righe di codice per la parte di elaborazione, compresa di unificazione e regole di tipaggio dei termini dell'elaboratore.

Per un totale di circa 2300 righe di codice.

Di quanto sviluppato, le parti riguardanti il *main loop*, la componente *language dependent* dell'*interactive theorem prover* e l'estensione con i termini dell'elaboratore risultano piuttosto stabili e non dovrebbero richiedere particolare *testing*. Viceversa, la componente che sicuramente richiede validazione è quella dell'unificazione, come anticipato.

L'uso di λProlog, con l'interprete ELPI, sembra essere chiaramente la strada giusta da percorrere, con l'auspicio che in futuro vengano intraprese non solo ulteriori rielaborazioni di quanto esposto, ma anche lo sviluppo di nuovi lavori che seguendo direzioni nuove in un ambito consolidato, possano portarlo a progredire.

Ringraziamenti

I miei più sentiti ringraziamenti vanno innanzitutto al mio relatore, prof. Claudio Sacerdoti Coen, che mi ha seguito con pazienza, rispondendo a ogni dubbio e curiosità, e senza il quale non avrei potuto completare questo lavoro.

Ringrazio inoltre Irene per avermi sempre supportato, per aver reso stupendi anche i momenti difficili e alla quale devo molto.

Infine, un ringraziamento va alla mia famiglia e amici per il sostegno in questi anni di studio.

Bibliografia

Abrahams, Paul (1963). «Machine Verification of Mathematical Proof». Tesi di dott. Massachussets Institute of Technology.

Asperti, Andrea (2009). *A survey on Interactive Theorem Proving.* Rapp. tecn. Mura Anteo Zamboni 7, 40127, Bologna, Italia: Dipartimento di Informatica, Università di Bologna.

Asperti, Andrea, Ferruccio Guidi, Claudio Sacerdoti Coen, Enrico Tassi e Stefano Zacchiroli (2004). «A Content Based Mathematical Search Engine: Whelp». In: *Proceedings of the 2004 International Conference on Types for Proofs and Programs.* TYPES'04. Jouy-en-Josas, France: Springer-Verlag, pp. 17–32. DOI: 10.1007/11617990_2. URL: http://dx.doi.org/10.1007/11617990_2.

Asperti, Andrea, Wilmer Ricciotti, Claudio Sacerdoti Coen e Enrico Tassi (2009). «Hints in Unification». In: *Theorem Proving in Higher Order Logics.* A cura di Stefan Berghofer, Tobias Nipkow, Christian Urban e Makarius Wenzel. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 84–98. ISBN: 978-3-642-03359-9.

— (2011). «The Matita Interactive Theorem Prover». In: *Proceedings of the 23rd International Conference on Automated Deduction.* CADE'11. Wrocław,

Poland: Springer-Verlag, pp. 64–69. ISBN: 978-3-642-22437-9. URL: http://dl.acm.org/citation.cfm?id=2032266.2032273.

Asperti, Andrea, Wilmer Ricciotti, Claudio Sacerdoti Coen e Enrico Tassi (2012). «A Bi-Directional Refinement Algorithm for the Calculus of (Co)Inductive Constructions». In: *Logical Methods in Computer Science* 8.1. DOI: 10.2168/LMCS-8(1:18)2012. URL: https://doi.org/10.2168/LMCS-8(1:18)2012.

Barendregt, Henk (1997). «The Impact of the Lambda Calculus in Logic and Computer Science». In: *BULLETIN OF SYMBOLIC LOGIC* 3, pp. 181–215.

Coen, Claudio e Enrico Tassi (2008). «Working with Mathematical Structures in Type Theory». In: *Proceedings of the 2007 International Conference on Types for Proofs and Programs*. TYPES'07. Cividale del Friuli, Italy: Springer-Verlag, pp. 157–172. URL: http://dl.acm.org/citation.cfm?id=1786134.1786145.

Coq developement team (2017). *The Coq proof assistant reference manual*. URL: https://coq.inria.fr/refman/.

Curry, H. B. (1934). «Functionality in Combinatory Logic». In: *Proceedings of the National Academy of Sciences* 20.11, pp. 584–590. ISSN: 0027-8424. DOI: 10.1073/pnas.20.11.584. eprint: http://www.pnas.org/content/20/11/584.full.pdf. URL: http://www.pnas.org/content/20/11/584.

de Bruijn, N. G. (1968). *Automath, a language for mathematics*. Rapp. tecn. 68-WSK-05. Department of Mathematics, Eindhoven University of Technology.

— (1970). «The mathematical language AUTOMATH, its usage, and some of its extensions». In: *Symposium on Automatic Demonstration*. A cura di

M. Laudet, D. Lacombe, L. Nolin e M. Schützenberger. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 29–61. ISBN: 978-3-540-36262-3.

de Moura, Leonardo Mendonça, Jeremy Avigad, Soonho Kong e Cody Roux (2015). «Elaboration in Dependent Type Theory». In: *CoRR* abs/1505.04324. arXiv: 1505.04324. URL: http://arxiv.org/abs/1505.04324.

Dunchev, Cvetan, Ferruccio Guidi, Claudio Sacerdoti Coen e Enrico Tassi (2015). «ELPI: fast, Embeddable, λProlog Interpreter». In: *Proceedings of LPAR*. Suva, Fiji. URL: https://hal.inria.fr/hal-01176856.

Dunchev, Cvetan, Claudio Sacerdoti Coen e Enrico Tassi (2016). «Implementing HOL in an Higher Order Logic Programming Language». In: *Proceedings of the Eleventh Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*. LFMTP '16. Porto, Portugal: ACM, 4:1–4:10. ISBN: 978-1-4503-4777-8. DOI: 10.1145/2966268.2966272. URL: http://doi.acm.org/10.1145/2966268.2966272.

Felty, Amy P. (1989). «Specifying and Implementing Theorem Provers in a Higher-order Logic Programming Language». AAI9015087. Tesi di dott. Philadelphia, PA, USA.

Felty, Amy P. e Dale Miller (1988). «Specifying Theorem Provers in a Higher-Order Logic Programming Language.» In: vol. 310, pp. 61–80. DOI: 10.1007/BFb0012823.

Fiori, Alberto e Claudio Sacerdoti Coen (2018). «Towards an Implementation in LambdaProlog of the Two Level Minimalist Foundation». In: *CICM 2018*. A cura di O. Hasan e F. Rabe. Hagenberg, Austria.

Frühwirth, Thom (1998). «Theory and practice of constraint handling rules». In: *The Journal of Logic Programming* 37.1, pp. 95–138. ISSN: 0743-1066.

DOI: https://doi.org/10.1016/S0743-1066(98)10005-5. URL: http://www.sciencedirect.com/science/article/pii/S0743106698100055.

Geuvers, Herman e Gueorgui I. Jojgov (2002). «Open Proofs and Open Terms: A Basis for Interactive Logic». In: *Computer Science Logic*. A cura di Julian Bradfield. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 537–552. ISBN: 978-3-540-45793-0.

Goldfarb, Warren D. (1981). «The undecidability of the second-order unification problem». In: *Theoretical Computer Science* 13.2, pp. 225–230. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(81)90040-2. URL: http://www.sciencedirect.com/science/article/pii/0304397581900402.

Gordon, Michael J. C., Robin Milner e Christopher P. Wadsworth (1979). *Edinburgh LCF: A Mechanised Logic of Computation*. Springer Verlag.

Guard, J. R., F. C. Oglesby, J. H. Bennett e L. G. Settle (1969). «Semi-Automated Mathematics». In: *J. ACM* 16.1, pp. 49–62. ISSN: 0004-5411. DOI: 10.1145/321495.321500. URL: http://doi.acm.org/10.1145/321495.321500.

Guidi, Ferruccio, Claudio Sacerdoti Coen e Enrico Tassi (2018). «Implementing Type Theory in Higher Order Constraint Logic Programming». working paper or preprint. URL: https://hal.inria.fr/hal-01410567.

Harper, Robert, Furio Honsell e Gordon Plotkin (1993). «A Framework for Defining Logics». In: *J. ACM* 40.1, pp. 143–184. ISSN: 0004-5411. DOI: 10.1145/138027.138060. URL: http://doi.acm.org/10.1145/138027.138060.

Harrison, John, Josef Urban e Freek Wiedijk (2014). «History of Interactive Theorem Proving». In: *Handbook of the History of Logic vol. 9 (Computational Logic)*. A cura di Jörg Siekmann. Elsevier, pp. 135–214.

Howard, W. A. (1980). «The formulae-as-types notion of construction.» In: *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*. A cura di J.P. Seldin e J.R. Hindley. L'originale circolava privatamente nel 1969. Academic Press, pp. 479–491.

Huet, Gerard P. (1973). «The undecidability of unification in third order logic». In: *Information and Control* 22.3, pp. 257–267. ISSN: 0019-9958. DOI: https://doi.org/10.1016/S0019-9958(73)90301-X. URL: http://www.sciencedirect.com/science/article/pii/S001999587390301X.

— (1975). «A unification algorithm for typed lambda-calculus». In: *Theoretical Computer Science* 1.1, pp. 27–57. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(75)90011-0. URL: http://www.sciencedirect.com/science/article/pii/0304397575900110.

List of long mathematical proofs (2018). *List of long mathematical proofs - Wikipedia, The Free Encyclopedia*. [Online; acceduta il 03 Dicembre 2018]. URL: https://en.wikipedia.org/wiki/List_of_long_mathematical_proofs.

MacKenzie, Donald (1995). «The Automation of Proof: A Historical and Sociological Exploration». In: *IEEE Ann. Hist. Comput.* 17.3, pp. 7–29. ISSN: 1058-6180. DOI: 10.1109/85.397057. URL: http://dx.doi.org/10.1109/85.397057.

Maietti, Maria Emilia (2009). «A minimalist two-level foundation for constructive mathematics». In: *Annals of Pure and Applied Logic* 160.3. Computation and Logic in the Real World: CiE 2007, pp. 319–354. ISSN: 0168-0072.

DOI: https://doi.org/10.1016/j.apal.2009.01.006. URL: http://www.sciencedirect.com/science/article/pii/S0168007209000104.

Martelli, A. e U. Montanari (1976). *Unification in linear time and space: a structured presentation.* Rapp. tecn. IEI-B76-16. Consiglio Nazionale delle Ricerche, Pisa.

Miller, Dale (1991a). «A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification». In: *Journal of Logic and Computation* 1.4, pp. 497–536. DOI: 10.1093/logcom/1.4.497. eprint: /oup/backfile/content_public/journal/logcom/1/4/10.1093/logcom/1.4.497/2/1-4-497.pdf. URL: http://dx.doi.org/10.1093/logcom/1.4.497.

— (1991b). «Unification of simply typed lambda-terms as logic programming». In: *In Eighth International Logic Programming Conference*. MIT Press, pp. 255–269.

— (1992). «Unification under a mixed prefix». In: *Journal of Symbolic Computation* 14.4, pp. 321–358. ISSN: 0747-7171. DOI: https://doi.org/10.1016/0747-7171(92)90011-R. URL: http://www.sciencedirect.com/science/article/pii/074771719290011R.

— (2018). «Mechanized metatheory revisited». In: *Journal of Automated Reasoning*. URL: https://hal.inria.fr/hal-01884210.

Miller, Dale e Gopalan Nadathur (1988). *An Overview of Lambda Prolog.* Rapp. tecn. Durham, NC, USA.

— (2012). *Programming with Higher-Order Logic*. Cambridge University Press. DOI: 10.1017/CBO9781139021326.

Newell, A. e H. Simon (1956). «The logic theory machine–A complex information processing system». In: *IRE Transactions on Information Theory* 2.3, pp. 61–79. ISSN: 0096-1000. DOI: 10.1109/TIT.1956.1056797.

Paterson, M.S. e M.N. Wegman (1978). «Linear unification». In: *Journal of Computer and System Sciences* 16.2, pp. 158–167. ISSN: 0022-0000. DOI: https://doi.org/10.1016/0022-0000(78)90043-0. URL: http://www.sciencedirect.com/science/article/pii/00220000789%2000430.

Robinson, J. A. (1965). «A Machine-Oriented Logic Based on the Resolution Principle». In: *J. ACM* 12.1, pp. 23–41. ISSN: 0004-5411. DOI: 10.1145/321250.321253. URL: http://doi.acm.org/10.1145/321250.321253.

Sacerdoti Coen, Claudio, Enrico Tassi e Stefano Zacchiroli (2007). «Tinycals: Step by Step Tacticals». In: *Electronic Notes in Theoretical Computer Science* 174.2. Proceedings of the 7th Workshop on User Interfaces for Theorem Provers (UITP 2006), pp. 125–142. ISSN: 1571-0661. DOI: https://doi.org/10.1016/j.entcs.2006.09.026. URL: http://www.sciencedirect.com/science/article/pii/S1571066107001740.

Saibi, Amokrane (1999). «Formalization of Mathematics in Type Theory. Generic tools of Modelisation and Demonstration. Application to Category Theory». Theses. Université Pierre et Marie Curie - Paris VI. URL: https://tel.archives-ouvertes.fr/tel-00523810.

Sozeau, Matthieu e Nicolas Oury (2008). «First-class type classes». In: *In Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar, editors, Theorem Proving in Higher Order Logics, 21st International Conference (TPHOLs '08), volume 5170 of LNCS*. Springer, pp. 278–293.

Sterling, Leon e Ehud Shapiro (1994). *The Art of Prolog (2Nd Ed.): Advanced Programming Techniques*. Cambridge, MA, USA: MIT Press. ISBN: 0-262-19338-8.

Ziliani, Beta e Matthieu Sozeau (2015). ≪A Unification Algorithm for Coq Featuring Universe Polymorphism and Overloading≫. In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. ICFP 2015. Vancouver, BC, Canada: ACM, pp. 179–191. ISBN: 978-1-4503-3669-7. DOI: 10.1145/2784731.2784751. URL: http://doi.acm.org/10.1145/2784731.2784751.