Studio sulle capacità espressive dei linguaggi di programmazione a paradigma logico nell'ambito del ragionamento automatico

Andrea Esposito

Area tematica— FORMAL MODELS, DATA ANALYSIS AND SCIENTIFIC COMPUTING.
ERC— PE1.
Keywords— Dimostrazione automatica, ragionamento automatico, linguaggi di programmazione, λProlog, programmazione a paradigma logico, ML, programmazione a paradigma funzionale, Teoria dei tipi, λcalcolo, unificazione.

1 Stato dell'arte

Allo stato attuale dello studio della logica applicata, ed in particolare nelle sue applicazioni nella dimostrazione automatica ed interattiva il paradigma di programmazione maggiormente scelto e preferito è quello funzionale. Di fatto alcuni fra i dimostratori automatici di maggior successo[1][2][3] sono scritti e sviluppati in linguaggi a paradigma funzionale, in particolare OCaml e gli altri linguaggi della famiglia ML potrebbero essere considerati come uno standard in questo senso. Un'alternativa meno diffusa ma promettente è quella dei linguaggi a paradigma logico, meno utilizzati e per questa ragione meno studiati e approfonditi.
Il linguaggio di programmazione λProlog [4], sviluppato da Dale e Gopalan Nadathur, ne è un esempio. Esso è basato su una versione intuizionistica della Teoria dei Tipi di Church [3] ed è a tutti gli effetti un sistema di ordine superiore che fornisce un framework molto potente, allargando le capacità espressive del Prolog classico ampliandone il linguaggio in modo da accettare anche le così dette *formule di Harrop ereditarie di ordine superiore* e fornito di meccanismi di binding che permettono, fra le altre cose, di gestire efficacemente la sostituzione di variabili vincolate nella sintassi e la così detta *binder mobilty* [17]. Il linguaggio è inoltre fornito di un sistema di tipizzazione polimorfico, ciò significa che termini e variabili possono avere più di un tipo, supportando sia un polimorfismo di tipo *universale* che *ad hoc*. Il sistema di tipi permette di organizzare i termini del linguaggio in una *gerarchia funzionale* e inoltre i tipi svolgono un importante ruolo nel rilevare errori nei programmi [19]. Il linguaggio si avvale dell'unificazione di ordine superiore per portare avanti la sua computazione, com'è noto [11] l'unificazione è un processo decidibile solo al prim'ordine, ed è sempre possibile, in questo contesto, trovare un unificante più generale (MGU). Spostando l'attenzione al second'ordine ci possiamo trovare con processi di unificazione che non terminano, Huet ha mostrato in [12] che anche provando a lavorare attorno a questo problema, considerando *insiemi di unificanti generali* (CSU), tali insiemi in generale non sono finiti, e non è nemmeno possibile dimostrare che tali infiniti unificanti non siano *non ridondanti*. Huet propone in [11] un algoritmo per la generazione di unificanti che nonostante le limitazioni teoriche citate, risulta molto efficace nella pratica [18]. Il linguaggio λProlog evita queste limitazioni avvalendosi di un sottoinsieme dei problemi di unificazione chiamato *Higher-Order pattern unification*. Limitando i termini unificabili ad una sottoclasse dei λ-termini chiamati appunto *pattern* (L_λ), Miller osserva che tali termini si comportano, nel processo di unificazione, come termini al

[1] `https://coq.inria.fr/`
[2] `https://www.cl.cam.ac.uk/~jrh13/hol-light/`
[3] `https://isabelle.in.tum.de/`
[4] `http://www.lix.polytechnique.fr/Labo/Dale.Miller/lProlog/`

prim'ordine e propone in [14] un algoritmo di unificazione che è sia decidibile che capace di trovare sempre l'unificante più generale. Altre implementazioni di λProlog restringono ulteriormente la loro attenzione a frammenti ancora più semplici, in modo da migliorare l'efficienza dell'unificazione, un esempio in questo senso è l'interprete ELPI[5] che limita l'attenzione dell'unificazione ad un sottoinsieme del linguaggio di λProlog chiamato *reduction-free fragment* ($\mathcal{L}_\lambda^\beta$) [4], che permette migliori prestazioni rispetto a Teyjus, l'implementazione di λProlog proposta di Miller stesso [6].

2 Obiettivi

L'obiettivo generale del lavoro è quello di fare uno studio approfondito sul linguaggio λProlog e più in generale sui linguaggi logici di ordine superiore, che si evolva possibilmente nella produzione di un dimostratore automatico scritto in uno di questi linguaggi. Tale studio avrà lo scopo di enucleare i maggiori punti di forza del paradigma di programmazione in questione, analizzando contestualmente i seguenti temi:

- Il confronto con altri paradigmi di programmazione, in particolare con quello funzionale, ampiamente utilizzato nello studio della dimostrazione automatica, operando confronti sia dal versante dell'implementazione della logica formale che dell'aspetto computazionale e di efficienza e del "design" dei programmi.
- Il tema del rapporto uomo/macchina e della maggiore leggibilità dei programmi scritti in linguaggi logici, che si prestano molto bene a codificare le regole logiche.
- La possibilità di ottenere una composizionalità del nucleo, che permetta di implementare diverse logiche diverse partendo da una base comune.
- Eventuali applicazioni della dimostrazione automatica nel contesto dell'apprendimento automatizzato e quindi dell'Intelligenza Artificiale.

Alcuni lavori nella direzione dell'implementazione di strategie di theorem proving in λProlog sono già presenti, Miller e Nadathur presentano un'implementazione di un calcolo a deduzione naturale in [16, p. 229] basandosi sui lavori di Felty [6, 5].

3 Metodologie

La fase di studio e quella di progettazione dovrebbe prendere circa un anno, in questo anno si dovrebbero approfondire la conoscenza del linguaggio λProlog, integrare nuove conoscenze su linguaggi di programmazione a paradigma funzionale , sviluppo del software e rafforzare le competenze in teoria della dimostrazione e logica matematica, al fine di avere una conoscenza più completa della materia da implementare. Implementare poi, con criterio, alcuni risultati classici di logica formale in λProlog, prendendo come punto di partenza il mio lavoro di tesi magistrale in cui una parte di questo lavoro è già stato svolto, un'altra direzione in cui si potrebbe proseguire sarebbe quella dell'implementazione della teoria dei tipi di Martin-Löf e delle sue estensioni, come il Calcolo delle Costruzioni induttive e il Fondamento Univalente.
Il resto del lavoro dovrebbe prendere circa un anno e mezzo/due anni in cui sarebbe da svolgere l'implementazione e l'ottimizzazione di tale progetto.

4 Risultati Attesi

La finalità del progetto è quella di scrivere un dimostratore automatico che sfrutti i vantaggi del paradigma di programmazione logico e utilizzi gli strumenti della logica di ordine superiore per rapp-

[5] `https://github.com/LPCIC/elpi`
[6] `http://teyjus.cs.umn.edu/`

resentare dimostrazioni e concetti di logica formale. Contestualmente sfruttando le capacità modulari di λProlog, tale progetto può diventare anche un vero e proprio Logical Framework che permetta anche una modularità che faciliti l'integrazione di nuove logiche e fornisca in generale una specificazione della logica in stile dichiarativo.
Un'altra direzione ancora potrebbe essere quella di usare λProlog per scrivere tools per altri dimostratori automatici già esistenti (e maggiormente usati), in modo da poter sviluppare estensioni per tali linguaggi in cui sia possibile manipolare facilmente espressioni che contengono *bindings*. Un lavoro del genere è già stato svolto per Coq [7], e si potrebbe realizzare lo stesso tipo di progetto di fatto per qualsiasi altro dimostratore.

References

[1] James Cheney. "Relating Nominal and Higher-Order Pattern Unification". In: *Proceedings of the 19th International Workshop on Unification (UNIF 2005)* (Jan. 2005).

[2] ALONZO CHURCH. *The Calculi of Lambda Conversion. (AM-6)*. Princeton University Press, 1941. ISBN: 9780691083940. URL: http://www.jstor.org/stable/j.ctt1b9x12d.

[3] Alonzo Church. "A formulation of the simple theory of types". In: *Journal of Symbolic Logic* 5.2 (1940), pp. 56–68. DOI: 10.2307/2266170.

[4] Cvetan Dunchev et al. "ELPI: fast, Embeddable, λProlog Interpreter". In: *Proceedings of LPAR*. Suva, Fiji, Nov. 2015. URL: https://hal.inria.fr/hal-01176856.

[5] Amy Felty. "Specifying and Implementing Theorem Provers in a Higher-Order Logic Programming, Language". PhD thesis. University of Pennsylvania, 1989.

[6] Amy Felty and Dale Miller. "Specifying theorem provers in a higher-order logic programming language". In: *9th International Conference on Automated Deduction*. Ed. by Ewing Lusk and Ross Overbeek. Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, pp. 61–80.

[7] D.M. Gabbay. "N-Prolog: An extension of prolog with hypothetical implication. II. Logical foundations, and negation as failure". In: *The Journal of Logic Programming* 2.4 (1985), pp. 251–283. ISSN: 0743-1066. DOI: https://doi.org/10.1016/S0743-1066(85)80003-0. URL: http://www.sciencedirect.com/science/article/pii/S0743106685800030.

[8] Mike Gordon. "From LCF to HOL: A Short History". In: *Proof, Language, and Interaction: Essays in Honour of Robin Milner*. Cambridge, MA, USA: MIT Press, 2000, pp. 169–185. ISBN: 0262161885.

[9] Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. "Implementing Type Theory in Higher Order Constraint Logic Programming". working paper or preprint. Nov. 2017. URL: https://hal.inria.fr/hal-01410567.

[10] John Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009. DOI: 10.1017/CBO9780511576430.

[11] G.P. Huet. "A unification algorithm for typed λ-calculus". In: *Theoretical Computer Science* 1.1 (1975), 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.

[12] Gérard Huet. "Unification in typed lambda calculus". In: *λ-Calculus and Computer Science Theory*. Ed. by C. Böhm. Berlin, Heidelberg: Springer Berlin Heidelberg, 1975, pp. 192–212. ISBN: 978-3-540-37944-7.

[7] https://github.com/LPCIC/coq-elpi

[13] L.Thorne McCarty. "Clausal intuitionistic logic I. fixed-point semantics". In: *The Journal of Logic Programming* 5.1 (1988), pp. 1–31. ISSN: 0743-1066. DOI: https://doi.org/10.1016/0743-1066(88)90005-2. URL: http://www.sciencedirect.com/science/article/pii/0743106688900052.

[14] Dale Miller. "A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification: Extended Abstract". In: *Proceedings of the 1989 Banff meeting on "Higher-Orders"*. Ed. by Graham M. Birtwistle. Banff, Canada, Sept. 1989.

[15] Dale Miller. "A logical analysis of modules in logic programming". In: *Journal of Logic Programming* 6.1-2 (Jan. 1989), pp. 79–108.

[16] Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, 2012. DOI: 10.1017/CBO9781139021326.

[17] Dale Miller and Alwen Tiu. "A proof theory for generic judgments". In: *ACM Trans. Comput. Log.* 6 (Oct. 2005), pp. 749–783. DOI: 10.1145/1094622.1094628.

[18] Tobias Nipkow. "Functional Unification of Higher-Order Patterns". In: *Proc. 8th IEEE Symp. Logic in Computer Science*. 1993, pp. 64–74.

[19] Frank Pfenning, ed. *Types in Logic Programming*. The MIT Press, 1992. ISBN: 0-262-16131-1.

[20] J. A. Robinson. "A Machine-Oriented Logic Based on the Resolution Principle". In: *J. ACM* 12.1 (Jan. 1965), pp. 23–41. ISSN: 0004-5411. DOI: 10.1145/321250.321253. URL: https://doi.org/10.1145/321250.321253.