Proceedings of the
Workshop on the

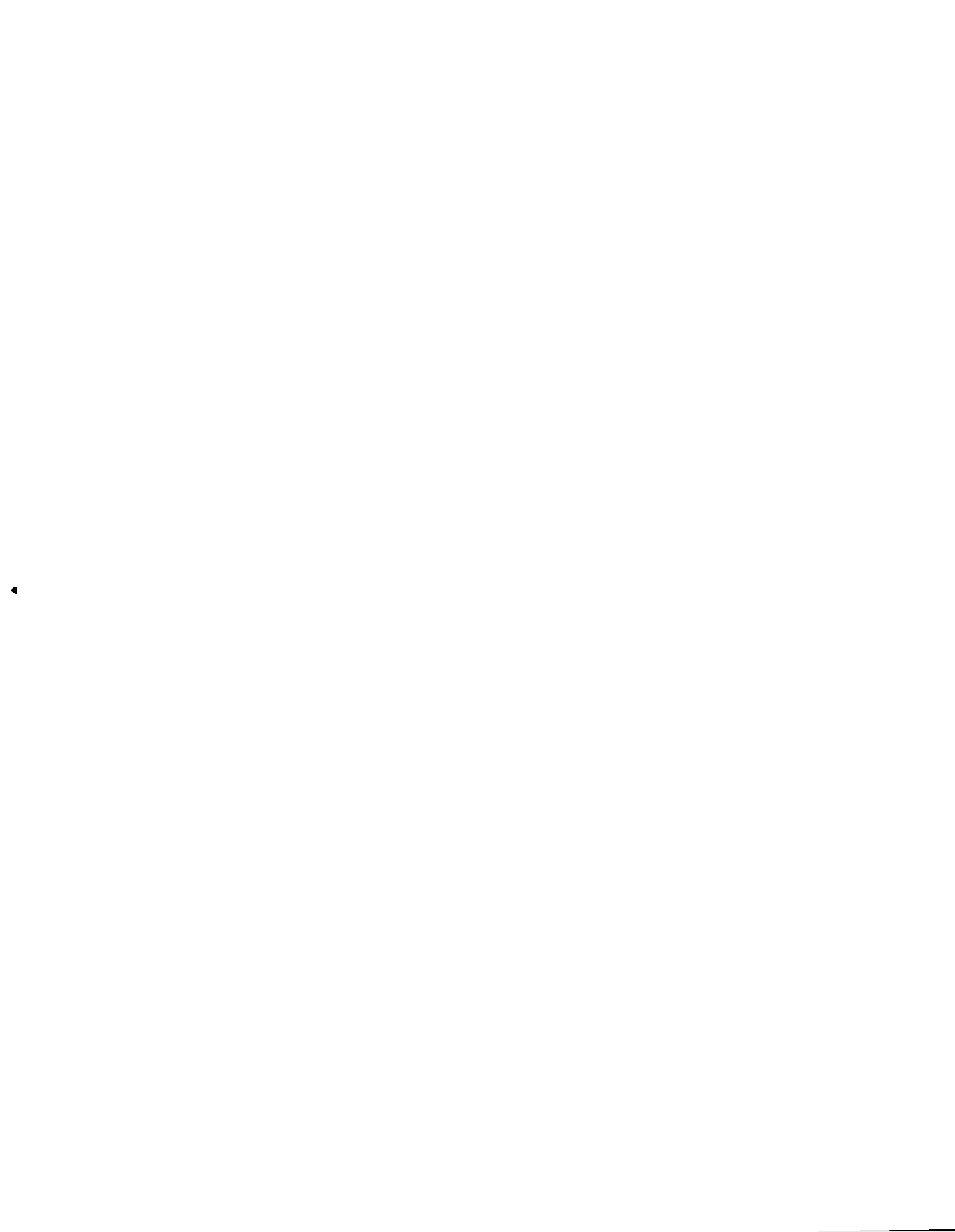
λProlog

Programming Language

31 July - 1 August 1992
University of Pennsylvania
Philadelphia, PA, USA

Sponsored by the
Institute for Research in
Cognitive Science, UPenn

Edited by Dale Miller
MS-CIS-92-86



Call for Papers and Participation

Workshop on the λProlog Programming Language

31 July - 1 August 1992 (Friday/Saturday)
University of Pennsylvania
Philadelphia, PA, USA

The expressiveness of logic programs can be greatly increased over first-order Horn clauses through a stronger emphasis on logical connectives and by admitting various forms of higher-order quantification. The logic of hereditary Harrop formulas and the notion of uniform proof have been developed to provide a foundation for more expressive logic programming languages. The lambda Prolog language is actively being developed on top of these foundational considerations. The rich logical foundations of lambda Prolog provides it with declarative approaches to modular programming, hypothetical reasoning, higher-order programming, polymorphic typing, and meta-programming. These aspects of lambda Prolog have made it valuable as a higher-level language for the specification and implementation of programs in numerous areas, including natural language, automated reasoning, program transformation, and databases.

This two day workshop will cover all aspects of the lambda Prolog language, including the following broad areas:

(1) Applications and programming techniques,
(2) Language design and definition,
(3) Implementations of interpreters and compilers, and
(4) Analysis of and extensions to the logical foundation.

Demonstrations of language implementation and applications are planned. A conference record containing submitted abstracts and papers will be made available at the workshop.

Deadlines: 30 April 1992 Submission deadline
22 May 1992 Notification of acceptance

Contributions: Submissions can be either abstracts of a minimum of 3 pages or full length papers. Persons interested in presenting system demonstrations or applications should send a two page description of their demo. Submissions will be judged by the organizing committee. If you are interested in attending this workshop but do not wish to submit a paper, please register your interest with the workshop chair.

Organizing Committee:

Elsa Gunter, AT&T Bell Labs
Dale Miller (chair), University of Pennsylvania
Gopalan Nadathur, Duke University
Frank Pfenning, Carnegie Mellon University

Sponsored by the Institute for Research in Cognitive Science, UPenn.

PREFACE

The first workshop on the λProlog language was held 31 July – 1 August 1992. Interest in λProlog has grown a great deal in the past several years. There is now active work in all areas of its theory, application, design, and implementation, including such topics as hypothetic reasoning, modular programming, proof theory, program transformation, natural language parsing and understanding, theorem proving, rewriting, generalization, compilation, and abstract machines. This workshop brought many of the people working on various aspects of λProlog together to discuss common problems and perspectives. This two day workshop attracted more than 30 attendees.

Robert Harper (Carnegie Mellon University) and Fernando Pereira (AT&T Bell Labs) kindly accepted to give invited talks. Harper spoke on "Modules for Elf" and Pereira spoke on "Semantic Interpretation as Higher-Order Deduction." Two computer systems were also demonstrated: the Prolog/Mali implementation of λProlog was demonstrated by Olivier Ridoux and the linear refinement of λProlog, Lolli, was demonstrated by Joshua Hodas. There were also 16 contributed papers, which are contained in these proceedings.

There is an electronic mailing list for discussions and announcements pertaining to λProlog and related topics. The current list contains more than 250 addresses. To be added to this list, send e-mail to `lprolog-request@cis.upenn.edu`.

I would like to thank the organizing and program committee — Elsa Gunter (AT&T Bell Labs), Gopalan Nadathur (Duke University), and Frank Pfenning (Carnegie Mellon University) — for their helped in designing the format of this workshop and for reading and reviewing all submitted papers. I would also like to thank Billie Holland for her help in local arrangements and with putting together this proceedings. Finally, I would like to thanks the Institute for Research in Cognitive Science at the University of Pennsylvania for providing the funds and facilities for holding this workshop.

Dale Miller
University of Pennsylvania
Philadelphia, PA, USA
December 1992

Contributed Papers

Model Theoretical Semantics for Higher-Order Horn Clause Programming
by Mino Bai, Syracuse University, NY, USA 1

The Architecture of an Implementation of λProlog: Prolog/Mali
by Pascal Brisset and Olivier Ridoux, IRISA/INRIA, France 41

Higher-Order Substitutions
by Dominic Duggan, University of Waterloo, ONT, Canada 65

Defining object-level parsers in λProlog
by Amy Felty, AT&T Bell Labs, NJ, USA 87

A Deductive Database View of Embedded Implications
by Burkhard Freitag, Technical University of Munich, Germany 101

From Context-Free to Definite-Clause Grammars
by Juergen Haas and Bharat Jayaraman, University of Buffula, NY, USA 113

Generalization at Higher Types
by Robert Hasker and Uday Reddy, University of Illinois, IL, USA 123

Implementing Higher-Order Algebraic Specifications
by Jan Heering, CWI, Amsterdam, The Netherlands 141

Lolli: An Extension to λProlog with Linear Logic Context Management
by Joshua Hodas, University of Pennsylvania, PA, USA 159

Some Kind of Magic for (a Restriction of) Lλambda
by Alain Hui Bon Hoa, INRIA Rocquencourt, France 169

An Instruction Set for Higher-Order Hereditary Harrop Formulas
by Keehang Kwon and Gopalan Nadathur, Duke University, NC, USA 195

Implementing the Module Construct in λProlog
by Keehang Kwon, Gopalan Nadathur, and Debra Sue Wilson,
Duke University, NC, USA 201

λProlog Implementation Of Ripple-Rewritting
by Chuck Liang, University of Pennsylvania, PA, USA 235

Searching for Inductive Proofs in Second-Order Intuitionistic Logic
by L. Thorne McCarty, Rutgers University, NJ, USA 243

An Empirical Study of the Runtime Behavior of Higher-Order Logic Programs
by Spiro Michaylov and Frank Pfenning, Carnegie Mellon Univ., PA, USA ... 257

A Proposal for Modules in λProlog
by Dale Miller, University of Pennsylvania, PA, USA......................... 273

The cover art work was painted by Shun-Wah Ma, Hong Kong.

LISTING OF ATTENDEES

Mino Bai
Syracuse University
School of Computer & Information Science
Center for Science and Technology
Syracuse, NY 13244-4100
mbai@top.cis.syr.edu

Michael Bukatin
Biosym Technologies, Inc.
23 W. Dewey Avenue
Wharton, NJ 07885
micheal@biocl.biosym.com

Wilfred Chen
Cornell University
Department of Computer Science
4141 Upson Hall
Ithaca, NY 14853
chen@cs.cornell.edu

Iliano Cervesato
University of Houston
Department of Computer Science
4800 Calhoun Road
Houston,TX 77204-3475
iliano@cs.uh.edu

Amy Felty
AT&T Bell Laboratories - 2A-425
600 Mountain Ave.
Murray Hill, NJ 07974
felty@research.att.com

Stacy Finkelstein
University of Pennsylania
Mathematics Department
Philadelphia, PA 19104-6389 USA
stacy@saul.cis.upenn.edu

Dr. Burkhard Freitag
Instutut fuer Informatik
Technische Universitaet Muenchen
Orleansstr. 34
D-8000 Muenchen 80
Germany
freitag@informatik.tu-muenchen.de

Vijay Gehlot
University of Glasgow
17 Lilybank Gardens
Glasgow G12 8QQ, Scotland (UK)
vijay@dcs.glasgow.ac.uk

Kannan Govindarajan
SUNY - Buffalo
Department of Computer Science
226 Bell Hall
Buffalo, NY 14260
govin-k@cs.buffalo.edu

Elsa L. Gunter
AT&T Bell Laboratories - 2A-432
600 Mountain Ave,
Murray Hill, NJ
elsa@research.att.com

Robert Harper
Carnegie Mellon University
School of Computer Science
Pittsburgh, PA 15213
rwh+@proof.ergo.cs.cmu.edu

Robert W. Hasker
University of Illinois
Department of Computer Science
1304 W. Springfield
Urbana, IL 61801
hasker@cs.uiuc.edu

Juergen Haas
SUNY - Buffalo
Department of Computer Science
226 Bell Hall
Buffalo, NY 14260
haas@cs.buffalo.edu

Jan Heering
CWI
Kruislaan 413
1098 SJ Amsterdam
The Netherlands
jan@cwi.nl

Joshua S. Hodas
University of Pennsylvania
Department of Computer & Information Science
200 South 33rd Street
Phila PA 19104
hodas@saul.cis.upenn.edu

Jonathan Hodgson
Saint Joseph's University
Department of Math/CSC
5600, City Avenue
Philadelphia. PA 19131
jhodgson@sju.edu

Alain Bon Hoa Hui
University of Pennsylvania
Department of Computer & Information Science
200 South 33rd Street
Philadelphia, PA 19104-6389
alain@saul.cis.upenn.edu

Bharat Jayaraman
SUNY - Buffalo
Department of Computer Science
226 Bell Hall
Buffalo, NY 14260
bharat@cs.buffalo.edu

Keehang Kwon
Duke University
Department of CPS
Durham, NC 27706
kwon@duke.cs.duke.edu

Chuck Liang
University of Pennsylvania
Department of Computer & Information Science
200 South 33rd Street
Philadelphia, PA 19104
liang@saul.cis.upenn.edu

L. Thorne McCarty
Rutgers University
Department of Computer Science
New Brunswick, NJ 08903
mccarty@cs.rutgers.edu

Raymond McDowell
Univ. of Pennsylvania
Computer & Information Science
200 South 33rd Street - Moore 370
mcdowell@saul.cis.upenn.edu

Spiro Michaylov
The Ohio State University
Department of Computer & Information Science
228 Bolz Hall
2036 Neil Avenue Mall
Columbus, OH 43210-1277
spiro@cs.cmu.edu

Dale Miller
University of Pennsylvania
Department of Computer & Information Science
Philadelphia, PA 19104-6389 USA
dale@cis.upenn.edu

Gopalan Nadathur
Duke University
Department of Computer Science
Durham, NC 27706
gopalan@cs.duke.edu

Fernando Pereira
AT&T Bell Labs - 2D-447
600 Mountain Ave.
Murray Hill, NJ 07974-0636
pereira@research.att.com

Olivier Ridoux
IRISA/INRIA
Campus Universitaire de Beaulieu
35042 RENNES Cedex FRANCE
ridoux@irisa.fr

Leon Shklyar
Rutgers University
Department of Computer Science
Bellcore, PYA 1E132
6 Corporate Place
Piscataway, NJ 08854
shklyar@cs.rutgers.edu

Debra Wilson
901 Chalk Level Rd.- Apt W7
Durham NC 27704
dsw@cs.duke.edu

General Model Theoretic Semantics and Negation as Failure in Higher-Order Logic Programming

Mino Bai [1]
School of Computer and Information Science
Syracuse University
Syracuse, New York 13244-4100, USA
mbai@top.cis.syr.edu

1 Abstract

We introduce model-theoretic semantics [6] for Higher-Order Horn logic programming language. We define general programs where the bodies of program clauses may contain negation symbol. We also define an interpreter for general programs. To derive a negative goal we need a negation as failure rule. For this, SLDNF-resolution with equality theory is also developed. We prove the soundness theorem analogous to Clark's fundamental theorem in [10].

2 Introduction

Many extended versions of Prolog are developed which incorporate higher-order features in logic programming languages to make programs more versatile and expressive [28, 8, 1]. In this paper, we build a model-theoretic semantics for a higher-order logic programming language which is suitable for describing declaratively operations of such programming language.

Church [9] introduced a simple theory of types as a system of higher-order logic. This system incorporated λ-notation in its particularly simple syntax which actually be viewed as a version of simply typed λ-calculus. Henkin first gave a semantics for Church's system based on general models. Domain members of a general model are truth values, individuals, and functions. Church's system was proved to be complete with respect to Henkin's semantics [15]. Andrews studied general models further in [3, 4, 5], and built a non-extensional model which is suitable under settings of resolution theorem proving [2]. The proof theory for this system is shown to have a close resemblance to that of first-order logic: there is, for example, a generalization to Herbrand theorem that holds for a variant of this system [22, 23].

λProlog [28] was the first language to show that higher-order logic could be used as the basis of a practical programming language. λProlog is based on typed λ-calculi which have their ultimate origin in Russel's method of stratifying sets to avoid the set theoretic paradoxes. One advantage of logic programs over coventional non-logic programs has been that they have simple declarative model-theoretic semantics. That is, in logic programs the least fixpoint is equal to least model, therefore it is associated to logical consequences and has a meaninful declarative interpretation. In

[1] *Address correspondence to* the author, School of Computer and Information Science, Center for Science and Technology/Fourth Floor, Syracuse University, Syracuse, New York 13244-4100, USA, *Telephone number of* the author, 315-443-2466

higher-order logic on which λProlog is based, compared to first-order case, it is extremely difficult to build an effective model-theoretic semantics. One of these difficulties is that the definition of satisfaction of formulas is mutually recursive with the process of evaluation of terms (see [15, 2, 3, 4, 5]). In first-order case, the model-theory is two level [19]. First we define a domain of individuals, and then define satisfaction wrt this domain. As a result of this in higher-order logic it is difficult to define $\mathsf{T}_{\mathcal{P}}$ operator for a logic program $\mathcal{P}$: In a definition of $\mathsf{T}_{\mathcal{P}}$ operator for a logic program $\mathcal{P}$, we consider a set of atomic propositions as an interpretation, and need a fixed domain without regard to interpretations. The second reason is that since higher-order logic programming languages are usually formulated in *non-extensional* form, we need a non-extensional model to describe properly such languages.

Henkin's general model semantics is extensional: i.e., if two objects in a model have the same extension, then they must be equal. Extensional models are very difficult to deal with, and unsuitable to describe a higher-order logic programming language like λProlog which contain a propositional type in its primitive set of types. For example, we can define a program $\mathcal{P}_1 = \{p(a) \leftarrow \top, q(a) \leftarrow \top, r(p(a)) \leftarrow \top\}$ in λProlog. Given program $\mathcal{P}_1$, the goal $r(p(a))$ will succeed in λProlog, but the goal $r(q(a))$ will fail, since the unification of $r(q(a))$ and $r(p(a))$ will simply fail. For any extensional model $\mathcal{M}$ for $\mathcal{P}_1$, $\mathcal{M}$ will assign the value $\mathbf{T}$ for $p(a)$ and $q(a)$. So $p(a) = q(a)$ is a logical consequence of $\mathcal{P}_1$. $\mathcal{M}$ will also assign the value $\mathbf{T}$ to $r(p(a))$, so the extension of the predicate which $\mathcal{M}$ will assign to r contains $\mathbf{T}$. Therefore $r(q(a))$ is a logical consequence of the program $\mathcal{P}_1$. Note that for this program the valuation of terms is mutually recursive with the satisfaction of formulas, since a formula can occur as an argument of predicate or functional symbols.

As shown above extensional models are difficult to define and unsuitable for higher-order logic programming. In this paper, we develop a non-extensional model where domain is independent from interpretations and build a fixed point semantics, and we prove the completeness of the interpreter in [26].

3 Higher-Order Horn Logic Programming Language

In this section we describe a higher-order logic programming language for which we build models in the later sections. For the exposition of our logic programming language $\mathcal{L}$ we will follow closely those in [28, 27].

The set $\mathcal{T}$ of types contains a collection $\mathcal{T}_0$ of primitive types and is closed under the formation of functional types: i.e., if $\alpha, \beta \in \mathcal{T}$, then $(\alpha \rightarrow \beta) \in \mathcal{T}$. The type constructor $\rightarrow$ associates to the right. The type $(\alpha \rightarrow \beta)$ is that of a function from objects of type α to objects of type β.

We introduce a very convenient notation from [29]. For each type symbol α, and each set S containing objects or expressions, we write S_α to denote the *set of things in S which are of type α*. We sometimes write $\{S_\alpha\}_\alpha$ to denote S. We can also define a *type assignment mapping* τ on the set S such that $\tau : S \rightarrow \mathcal{T}$ and for all $s \in S$, $\tau(s) = \alpha$ if $s \in S_\alpha$.

Let S, T, T_1, T_2 be sets. Given a mapping $f : S \rightarrow T$, $a \in S$, and $b \in T$, let $f[b/a]$ be that mapping $f' : S \rightarrow T$ such that for $f'a = b$ and $f'c = fc$ for all $c \neq a$. Let b be an element in $T_1 \times T_2$, then b^1 and b^2 are the first and second components of b, so $b = \langle b^1, b^2 \rangle$. If f is a mapping

whose values are in $T_1 \times T_2$, let f^1 and f^2 be mappings with the same domain as f defined so that for any argument t, $f^i t = (ft)^i$ for $i = 1, 2$. Thus $ft = \langle f^1 t, f^2 t\rangle$. If $f : S \to T$ is a mapping, then we say that f is *type consistent* if for all $s \in S$, $\tau(f(s)) = \tau(s)$. If $f : S \to T_1 \times T_2$, then we say that f is *type consistent* if f^1 and f^2 are type consistent. For each integer $n \in \omega$, we write $[n]$ for the set $\{1, \cdots, n\}$.

We assume that there are denumerably many variables and constants of each type. Let the set of variables and constants be Δ and Σ, respectively. *Simply typed λ-terms* are built up in the usual fashion from these typed constants and variables via abstraction and application. Our *well formed terms* (wfts) are simply typed λ-terms. We, as usual, can define the set $T(\Sigma)$ of all wfts by giving the definition of the set $T(\Sigma)_\alpha$ of wfts of type α by induction.

It is assumed that the reader is familiar with most of basic notions and definitions such as bound, free variables, closed terms (c-terms), substitution and λ-conversion for this language; only a few are reviewed here. Letters $f_\alpha, s_\alpha, t_\alpha, \cdots$, will be used as syntactical variables of wfts of type α. Type subscript symbols may be omitted when context indicates what they should be or irrelevant to discussion. By Church-Rosser theorem [7], a λ-normal wfts of a wft is unique upto a renaming of variables. For most part we shall be satisfied with any of these normal forms corresponding to a wft t, and we shall write $\lambda norm(t)$ to denote such a form. In certain situations we shall need to talk about a unique normal form and, in such cases, we shall use $\rho(t)$ to designate what we shall call the *principal normal* or *ρ-normal* form of t; i.e. ρ is a mapping from wfts to λ-normal terms. There are several schemes that may be used to pick a representative of the α-equivalence classes of λ-normal terms and the one implicitly assumed here is that of [2].

So far we have introduced λ-term structures and operations on λ-terms. We can introduce logic into λ-term structures by including o, a type for propositions, amongst the set of primitive types $\mathcal{T}_0$, and requiring that the collection Σ of constants contain the following *logical* constants: $\wedge$ and $\vee$ of type $o \to o \to o$; $\top$ of type o; and for every type α, $\exists_\alpha$ of type $(\alpha \to o) \to o$. The constants in Σ other than $\wedge, \vee, \exists$ and $\top$ are called as *non-logical* constants. A type will be called a *predicate type* if it is a type of the form $\alpha_1 \to \cdots \alpha_n \to o$, or a *non-predicate type* otherwise. We let $\Pi \subseteq \Sigma$ be the *set of predicate constants*. Expression of the form $\exists(\lambda x G)$ will be abbreviated by $\exists x G$.

Terms of type o are referred to as *goal formula*. The λ-normal form of a goal formula consists, at the outermost level, of a sequence of applications, and the leftmost symbol in this sequence is called its *top level symbol*. We shall have use for the structure of λ-normal formulas that is described below. A goal formula is said to be an *atom* (*atomic*) if its leftmost symbol that is not a bracket is either a predicate variable or constant. A λ-normal goal formula G, then, has the following inductive characterization: (a) it is $\top$, (b) it is an atom, (c) it is $G_1 \wedge G_2$ or $G_1 \vee G_2$, where G_1 and G_2 are λ-normal goal formulas, or (d) it is $\exists x G$, where G is a λ-normal goal formula.

Now we identify the formulas that we call higher-order definite clauses, goal formula, and equations. Let $\mathcal{G}$ be the collection of all λ-normal goal formulas. An *atom* is an atomic goal formula A. A *rigid atom* is an atom A_r that has a predicate constant as its head. An atom is thus a formula of the form $pt_1 \cdots t_n$ where $\gamma = \alpha_1, \cdots, \alpha_n \to o$, p is a predicate constant$_\gamma$ or variable$_\gamma$, and, for each $i \in [n]$, t_i is a λ-normal term$_{\alpha_i}$, it is a rigid atom just in case p is a constant. Sometimes we write $p(t_1, \cdots, t_n)$ or $p(\bar{t})$ for the above atom. Let G be an arbitrary goal formula and A_r be any rigid atom. Let a formula C be of the form $A_r \leftarrow G$. Then C is a *(higher-order)*

definite clause. Let $s_\alpha, t_\alpha \in T(\Sigma)$. Then, as usual, an *equation* e is of the form $s_\alpha = t_\alpha$, and an *extensional equation* is of the form $s_\alpha \equiv t_\alpha$. Let $\mathcal{D}ef$ be the set of all definite clauses. Then given the collection Σ of constants, our *logic programming language* $\mathcal{L} = \mathcal{L}(\Sigma)$ is completely determined as the triple $\langle T(\Sigma), \mathcal{G}, \mathcal{D}ef\rangle$. A formula F in a language $\mathcal{L}$ is a goal formula, or a definite clause, or an equation. We refer a set $\mathcal{P}$ of formulas from $\mathcal{D}ef$ as a *higher-order definite logic program.* As usual, variables in definite clauses are implicitly universally quantified. Note that in the above definition all wfts in $T(\Sigma)$ do not contain such symbols as $=, \equiv, \leftarrow$, hence a goal formula G and s_α and t_α in an equation $s_\alpha = t_\alpha$ do not contain those symbols.

We say that a predicate symbol p *occurs extensionally in* a goal formula G if (a) G is $p(\bar{t})$, or (b) G is $G_1 \wedge G_2$ or $G_1 \vee G_2$, or and p occurs extensionally in G_1 or G_2, or (c) G is $\exists x G_1$, and p occurs extensionally in G_1. In following sections, we will define semantics for λProlog. We will take advantage of the following situation: Since logic programs compute extensions of predicates, and relations between arguments of predicate symbols constitute extensions of predicates, we don't need extensions of terms until we meet extensional occurrences of predicate symbols in the definition of satisfaction of formulas.

4 General Model Theoretic Semantics

In this Section we build model-theoretic semantics for the language $\mathcal{L}$. As introduced in Section 1 we need a non-extensional model to prove that a resolution system in type theory is complete. The model in [2] is in a sense non-extensional. But it doesn't provide an adequate notion of "general" non-extensional model for our purpose: Domain is defined by indexing extension of the element in it by wfts. The indexed entity like $\langle t, \mathbf{p}\rangle$ is called a V-complexe where V is a truth value evaluation of formulas. So only one kind of domain is used in [2], since the set of all wfts is predetermined given a language $\mathcal{L}$. In [2], in order to define the domain of interpretation we need a semivaluation function V, as above, which evaluates proposional formulas to **T** or **F**. The definition of domain or the evaluation of terms is mutually recursive with the definition of evaluation of formulas.

Now we generalize Andrews model to a model where we index the extension by an element from a general domain which we call frame. ¿From this model we build a model where the definition of domain is independent from the definition of satisfaction. These two models will be shown to be isomorphic and elementarily equivalent in the sense that the sets of valid sentences in each semantics are same. Since our language $\mathcal{L}$ is based on λ-calculus and application is a basic operation of the λ-calculus, any model of $\mathcal{L}$ should be an applicative structure which is a λ-model.

Definition Let A be a set and $\cdot$ a binary operation over A such that for all $\alpha, \beta \in \mathcal{T}$, for all $a \in A_{\alpha\rightarrow\beta}, b \in A_\alpha$, $a \cdot b$ is an element in A_β. Then $\mathcal{A} = \langle A, \cdot\rangle$ is said to be an *applicative structure.* An *assignment into* a set A is a type consistent mapping $\varphi : \Delta \rightarrow A$. A *λ-model* is a triple $\langle A, \cdot, ||\cdot||\rangle$ such that $\langle A, \cdot\rangle$ is an applicative structure and $||\cdot||$ a binary function such that for each assignment φ into A and term t_α, $||t_\alpha||_\varphi \in A_\alpha$, and for all terms $f \in T(\Sigma)_{\alpha\rightarrow\beta}$ and $t \in T(\Sigma)_\alpha$, $||ft||_\varphi = ||f||_\varphi \cdot ||t||_\varphi$, and for all term t and $a \in A_\alpha$, $||\lambda x_\alpha t||_\varphi \cdot a = ||t||_{\varphi[a/x_\alpha]}$. We call the function $||\cdot||$ a *valuation function* in A. □

A *frame* is a nonempty set D of objects each of which is assigned a type symbol from the set $\mathcal{T}$ in such a way that every object in $D_{\alpha\rightarrow\beta}$ is a function from D_α to D_β for all type symbols α and

β. A *pre-interpretation* $\mathcal{F}$ of the language $\mathcal{L}$ is a pair $\langle D, J\rangle$ where D is a frame, and J is a type consistent mapping in $\Sigma \to D$. An assignment into a pre-interpretation is an assignment into the frame of the pre-interpretation. Note that $D_{\alpha\to\beta}$ is some collection of functions mapping D_α into D_β, i.e. $D_{\alpha\to\beta} \subseteq D_\alpha \to D_\beta$. A pre-interpretation $\mathcal{F} = \langle D, J\rangle$ is said to be *general* iff there is a binary function $\mathsf{V}^{\mathcal{F}} = \mathsf{V}$ such that for each assignment φ and term t_α, $\mathsf{V}_\varphi t_\alpha \in D_\alpha$, and the following conditions are satisfied for each assignment φ and all terms: (a) if $x \in \Delta$, then $\mathsf{V}_\varphi x = \varphi x$. (b) if $c \in \Sigma$, then $\mathsf{V}_\varphi c = Jc$. (c) $\mathsf{V}_\varphi(ft) = (\mathsf{V}_\varphi f)\mathsf{V}_\varphi t$ (the value of the function $\mathsf{V}_\varphi f$ at the argument $\mathsf{V}_\varphi t$). (d) $\mathsf{V}_\varphi(\lambda x_\alpha t_\beta) = \boldsymbol{\lambda} d \in D_\alpha \cdot \mathsf{V}_{\varphi[d/x]}t_\beta$ i.e. that function from D_α into D_β whose value for each argument $d \in D_\alpha$ is $\mathsf{V}_{\varphi[d/x]}t_\beta$.

If a pre-interpretation $\mathcal{F}$ is general, the function $\mathsf{V}^{\mathcal{F}}$ is uniquely determined. We can prove this by induction on the definition of terms. We call the unique function $\mathsf{V}^{\mathcal{F}}$ the *intentional valuation function* of terms in the pre-interpretation $\mathcal{F}$. $\mathsf{V}^{\mathcal{F}}_\varphi t$ is called the *intention* of t in $\mathcal{F}$ wrt φ. We sometimes write $\mathsf{V}^{\mathcal{F}}_\varphi$ as V_φ, as $\mathsf{V}^{\mathcal{F}}$, or as V, when pre-interpretation or assignment is clear from context, or irrelevant. It is clear that if t is a c-term, then $\mathsf{V}^{\mathcal{F}}t$ may be considered meaningful without regard to any assignment. In this case, $\mathsf{V}^{\mathcal{F}}t$ is called the *intention* of t in $\mathcal{F}$ and written as t'. Obviously for a general frame D, $\langle D, \cdot, \mathsf{V}\rangle$ where $\cdot$ is interpreted as a functional application is a λ-model, but in a pre-interpretation logic symbols such as logical operators and predicate constants are not fully interpreted. So we call it a pre-interpretation.

Now we will give interpretations to logical symbols, after discussing a few constructions of posets. Any non-empty set A can be considered a poset under the identity relation where $x \subseteq_A y$ iff $x = y$. We call this type of poset *discrete*. Let P_1 and P_2 be disjoint posets. $P_1 \cup P_2$ is a poset $P = P_1 \cup P_2$ such that for all $x, y \in P$, $x \subseteq_P y$ if $x \subseteq_{P_1} y$ or $x \subseteq_{P_2} y$. $P_1 \times P_2$ is a poset $P = P_1 \times P_2$ where for all $x, y \in P$, $x \subseteq_P y$ if $x^1 \subseteq_{P_1} y^1$ and $x^2 \subseteq_{P_2} y^2$. Let S be a set, and P a poset. $S \to P$ is a poset F such that for all $f, g \in F$, $f \subseteq_F g$ if for all $s \in S$, $f(s) \subseteq_P g(s)$. Let $\mathcal{B}$ be the set of boolean values $\mathbf{T}$ and $\mathbf{F}$ where $\mathbf{F} \subseteq_{\mathcal{B}} \mathbf{T}$. We shall write $\vee$ and $\wedge$ for $\sqcup_{\mathcal{B}}$ and $\sqcap_{\mathcal{B}}$, respectively. Let A be a set. We can consider A a discrete poset. A *predicate* P *over* A *of type* $\alpha_1, \cdots, \alpha_n \to o$ is a mapping in $A_{\alpha_1} \times \cdots \times A_{\alpha_n} \to \mathcal{B}$, or equivalently a subset of $A_{\alpha_1} \times \cdots \times A_{\alpha_n}$. And we consider truth values $\mathbf{T}$ and $\mathbf{F}$ as *null-ary* predicates over A of type $() \to o$ such that $\mathbf{T}() \equiv \mathbf{T}$ and $\mathbf{F}() \equiv \mathbf{F}$, respectively. More generally, we define predicates $\mathbf{T}^A_{\alpha_1,\cdots,\alpha_n}$ for each list $\alpha_1, \cdots, \alpha_n$ of types where $n \geq 0$ as $A_{\alpha_1} \times \cdots \times A_{\alpha_n}$. We write $\Phi(A)$ for the *set of all predicates over* A. Given two predicates $P, Q \in \Phi(A)$, it is obvious that $P \subseteq Q$ if P and Q are of same type and P is a subset of Q.

Definition Let D be a frame. A *semivaluation* of D is a function V with domain D_o and range the set $\mathcal{B}$ of truth values such that the following properties hold: for all $c_o, d_o, f_{\alpha\to o} \in D$, (a) $V(\top') = \mathbf{T}$. (b) $V(\vee' c_o d_o) = V(c_o) \vee V(d_o)$. (c) $V(\wedge' c_o d_o) = V(c_o) \wedge V(d_o)$. (d) $V(\exists'_\alpha f_{\alpha\to o}) = \mathbf{T}$ iff there is some $e \in D_\alpha$ such that $V(f_{\alpha\to o}e) = \mathbf{T}$. Given a frame D and a semivaluation V of D, we define the *set* $\mathcal{D}$ *of* V*-complexes based on* D as follows: For each type γ we define the set $\mathcal{D}_\gamma$ of V-complexes$_\gamma$ and one-one onto mapping $\kappa_\gamma : D_\gamma \to \mathcal{D}_\gamma$ as follows by induction on γ: (a) $\mathcal{D}_o = \{\langle d, Vd\rangle : d \in D_o\}$. For $d \in D_o$, $\kappa_o d = \langle d, Vd\rangle$. (b) When $\alpha \in \mathcal{T}_0 - \{o\}$, $\mathcal{D}_\alpha = \{\langle d, d\rangle : d \in D_\alpha\}$. For $d \in D_\alpha$, $\kappa_\alpha d = \langle d, d\rangle$. (c) $\mathcal{D}_{\alpha\to\beta} = \{\langle f, \kappa_\alpha^{-1} \circ f \circ \kappa_\beta\rangle : f \in D_{\alpha\to\beta}\}$. For $f \in D_{\alpha\to\beta}$, $\kappa_{\alpha\to\beta} f = \langle f, \kappa_\alpha^{-1} \circ f \circ \kappa_\beta\rangle$. We say that $\mathcal{D}$ is the set of V-complexes *based on* D. We can also introduce one-one onto mapping $\kappa : D \to \mathcal{D}$ such that for $\alpha \in \mathcal{T}, d \in D_\alpha, \kappa d = \kappa_\alpha d$, and function $\boldsymbol{v}$ whose domain is D such that for $d \in D_\alpha$, $\boldsymbol{v}(d) = (\kappa d)^2$. □

Now it is easy to see that (a) if $f \in D_{\alpha\to\beta}$, then $v(f) : \mathcal{D}_\alpha \to \mathcal{D}_\beta$, (b) for $a \in \mathcal{D}_\alpha$, $v(f)a = \langle fa^1, v(fa^1)\rangle$, and (c) $\mathcal{D} = \{\langle d, v(d)\rangle : d \in D\}$. And for any $a \in \mathcal{D}$, $\kappa a^1 = a$, and for any mapping χ whose values are in $\mathcal{D}$, $\chi^1 \circ \kappa = \chi$. Let $\mathcal{D}$ be a set of V-complexes. Then we define the ***applicative operation*** $\star$ of type $(\alpha \to \beta), \alpha \to \beta$: For $a \in \mathcal{D}_{\alpha\to\beta}$ and $b \in \mathcal{D}_\alpha$, $a \star b$ is defined to be $a^2 b$. The operation $\star$ is left associative. Let $a \in \mathcal{D}_{\alpha_1,\cdots,\alpha_n\to\beta}$ and $b_i \in \mathcal{D}_{\alpha_i}$ for $i \in [n]$. Then by definition of $\mathcal{D}$ it is easy to see that $a \star b_1 \star \cdots \star b_n \in \mathcal{D}_\beta$. Moreover, $\langle \mathcal{D}, \star\rangle$ is an applicative structure and for all $f \in D_{\alpha\to\beta}, d \in D_\alpha$, $(\kappa f) \star (\kappa d) = \kappa(fd)$.

Definition Let $\mathcal{D}$ be a set of V-complexes. We can define a binary mapping $\mathcal{V}$ such that for all assignment φ into $\mathcal{D}$, $\mathcal{V}_\varphi : T(\Sigma) \to \mathcal{D}$, and for all $t \in T(\Sigma)$, $\mathcal{V}^1_\varphi t = \mathsf{V}_{\varphi^1} t$. □

Let φ be an assignment into D. Then for all term t, $\kappa \mathsf{V}_\varphi t = \mathcal{V}_{\varphi\circ\kappa} t$. If φ is an assignment into $\mathcal{D}$, then for $a \in \mathcal{D}_\alpha$, $\mathcal{V}_\varphi(\lambda x_\alpha t) \star a = \mathcal{V}_{\varphi[a/x_\alpha]} t$. If D be a general frame and $\mathcal{D}$ a set of V-complexes, then there is the unique $\mathcal{V}$ satisfying that for all $t_\gamma \in T(\Sigma)$ and assignment φ into $\mathcal{D}$, $\mathcal{V}_\varphi t_\gamma \in \mathcal{D}_\gamma$, since the function V is unique. Therefore $\langle \mathcal{D}, \star, \mathcal{V}\rangle$ is a λ-model.

Now we want to define a notion of extension of a V-complex in the usual mathematical sense: e.g., if $a \in \mathcal{D}_{\alpha_1,\cdots,\alpha_n\to o}$, then we want the extension of a to be a predicate over D.

Definition Given a frame D, we define a *primitive extensional domain* E_α for $\alpha \in \mathcal{T}_0$: (a) $E_o = \mathcal{B}$. (b) $E_\alpha = D_\alpha$ for $\alpha \in \mathcal{T}_0 - \{o\}$. Given an $a \in \mathcal{D}_{\alpha_1,\cdots,\alpha_n\to\beta}$ where $n \geq 0$ and $\beta \in \mathcal{T}_0$, we define a mapping $a^\odot$ in $D_{\alpha_1} \to \cdots \to D_{\alpha_n} \to E_\beta$ by induction on n: (a) When $n = 0$, $a^\odot = a^2$. (b) When $n > 0$, $a^\odot = \lambda d_1 \in D_{\alpha_1} \cdot (a \star \kappa d_1)^\odot$.

We call a^1 the *intention* of a, and $a^\odot$ the *extension* of a. □

Let $a \in \mathcal{D}_{\alpha_1,\cdots,\alpha_n\to\beta}$ where $n > 0$ and $\beta \in \mathcal{T}_0$. Then (a) for all $d_i \in D_{\alpha_i}$, $i \in [n]$, $a^\odot d_1 \cdots d_n = (a \star \kappa d_1 \star \cdots \kappa d_n)^2$, (b) If $\beta \in \mathcal{T}_0 - \{o\}$, then $a^\odot = a^1$. We can show this by induction on n.

Definition [2] Let $\mathcal{F} = \langle D, J\rangle$ be a general pre-interpretation and V a semivaluation of D. An ***$\mathcal{L}$-structure*** $\mathcal{A}$ is a pair $\langle \mathcal{D}, J\rangle$ such that $\mathcal{D}$ is a set of V-complexes based on D. We say that $\mathcal{A}$ is ***based on*** $\mathcal{F}$ or ***on*** D. An ***assignment*** φ ***into*** $\mathcal{A}$ is an assignment into $\mathcal{D}$. When F is a formula in $\mathcal{L}$, we write $\mathcal{A}{\models}F[\varphi]$ to say that $\mathcal{A}$ *satisfies* F wrt φ. (a) When $s_\alpha, t_\alpha \in T(\Sigma)$, $\mathcal{A}{\models}s_\alpha = t_\alpha[\varphi]$ iff $\mathcal{V}_\varphi s_\alpha = \mathcal{V}_\varphi t_\alpha$, $\mathcal{A}{\models}s_\alpha \equiv t_\alpha[\varphi]$ iff $(\mathcal{V}_\varphi s_\alpha)^\odot = (\mathcal{V}_\varphi t_\alpha)^\odot$. (b) When G is a goal formula, $\mathcal{A}{\models}G[\varphi]$ iff $\mathcal{V}^2_\varphi G = \mathbf{T}$. (c) When $A \leftarrow G$ is a definite clause, $\mathcal{A}{\models}A \leftarrow G[\varphi]$ iff $\mathcal{A}{\models}A[\varphi]$ whenever $\mathcal{A}{\models}G[\varphi]$. We write $\mathcal{A}{\models}F$ to say that a formula F is *valid* in $\mathcal{A}$ if $\mathcal{A}{\models}F[\varphi]$ for all assignments φ into $\mathcal{A}$. Given a set of definite clause $\mathcal{P}$, we say that $\mathcal{A}$ is a *model* or *D-model* for $\mathcal{P}$, and write $\mathcal{A}{\models}\mathcal{P}$, if each definite clause in $\mathcal{P}$ is valid in $\mathcal{A}$. Given a closed goal formula G, we say that G is a ***logical consequence*** of $\mathcal{P}$, and write $\mathcal{P}{\models}G$ if G is valid in all models of $\mathcal{P}$. □

Definition Let D be a general frame and S a subset of D_o. Then S is ***upward saturated*** if a) $\top' \in S$, b) $c \in S$ implies $\vee' cd, \vee' dc \in S$ for $d \in D_o$, c) $c, d \in S$ implies $\wedge' cd \in S$, and d) $f_{\alpha\to o} d_\alpha \in S$ implies $\exists'_\alpha f_{\alpha\to o} \in S$. □

Let $S \subseteq D_o$. Then there is a smallest upward saturated set extending S. Let $\mathcal{C}$ be the collection of upward saturated set extending S. $\mathcal{C}$ is not empty, since $D_o \in \mathcal{C}$. So $\cap\mathcal{C}$ exists. It is easy to check that it is upward saturated. It fulfills the other considerations, by definition. The smallest upward saturated set extending S is called the *upward saturated closure* of S, and is denoted as S^U.

[2] Note that in this definition the symbol for satisfaction in $\mathcal{A}$ is the small $\models$. The normal size $\models$ is used for another definition of satisfaction which is defined later in this paper.

If $S \subseteq D_o$ we can always find by the above method an extension of S which is saturated. The above definition is certainly simple, but it is unsatisfactory on several grounds. For example, it does not make explicit how the elements of the closure of S are generated from the elements of S. ¿From this reason we give a more constructive definition, involving restricted set-theoretic methods.

Definition Let $S \subseteq D_o$. An *elementary S-derivation* is a sequence $c^1, \cdots, c^m$, $m \geq 1$, of elements from D_o, where for each $i \in [m]$, at least one of the following conditions is satisfied: (a) $c^i = \top'$. (b) $c^i \in S$. (c) There is a $j < i$ such that c^i is either $\vee' c^j d$ or $\vee' d c^j$ for some $d \in D_o$. (d) There are $j, k < i$ such that $c^i = \wedge' c^j c^k$. (e) There are $j < i$ and $f \in D_{\alpha \to o}$ such that $c^j = fd$ for some $d \in D_\alpha$ and $c^i = \exists'_\alpha f$. □

Note that if $c^1, \cdots, c^m$ and $d^1, \cdots, d^n$ are two elementary S-derivations, then the concatenation $c^1, \cdots, c^m, d^1, \cdots, d^n$ is also an elementary S-derivation. Furthermore, a nonempty initial segment of an elementary S-derivation is again an elementary S-derivation. An element $d \in D_o$ is *elementary S-derivable* if there is an elementary S-derivation $c^1, \cdots, c^m$ where $c^m = d$. This is equivalent to requiring that d be an element (not necessarily the last) in some elementary S-derivation. The set of all $d \in D_o$ that are elementary S-derivable is denoted by $\mathsf{E}(S)$. We shall show that $\mathsf{E}(S)$ is the upward closure of S referred to above.

Theorem 4.1 *Let $S \subseteq D_o$. Then: (a) $S \subseteq \mathsf{E}(S)$. (b) $\mathsf{E}(S)$ is upward saturated. (c) If $S \subseteq S'$ and S' is upward saturated, then $\mathsf{E}(S) \subseteq S'$. (d) $S^U = \mathsf{E}(S)$.*

Proof The proofs of (a) and (b) are obvious. (c) Let $S \subseteq S'$ and S' be upward saturated. We prove by induction on m that whenever $c^1, \cdots, c^m$ is an elementary S-derivation then $c^i \in S'$ for $i \in [m]$. When $m = 1$. it is clear. If the property is true for m, and $c^1, \cdots, c^m, c^{m+1}$ is an elmentary S-derivation, then by IH we have that $c^i \in S'$ for $i \in [m]$. Furthermore c^{m+1} is $\top'$, or a $c \in S \subseteq S'$, or it is obtained by one of the defining rules from the elements in S'. In all cases it is easy to see, by IH and definition of upward saturatedness, that $c^{m+1} \in S'$. □

Definition Let $\langle D, J \rangle$ be a general pre-interpretation. Then we write $\Pi(D)$ for the *D-base* which is defined to be the set $\{p(a_1, \cdots, a_n) : p \in \Pi_{\alpha_1, \cdots, \alpha_n \to o}$ and $a_i \in D_{\alpha_i}$ for all $i \in [n]\}$. □

A subset $\mathcal{K}$ of $\Pi(D)$ induces a unique mapping $I_\mathcal{K}$ in $\Pi \to \Phi(D)$ as follows: for all $\overline{d} \in D$, $\langle \overline{d} \rangle \in I_\mathcal{K}(p)$ iff $p(\overline{d}) \in \mathcal{K}$. Let $\mathcal{K}_1 \subseteq \mathcal{K}_2 \subseteq \Pi(D)$, then it is easy to see that $I_{\mathcal{K}_1} \subseteq_{\Pi \to \Phi(D)} I_{\mathcal{K}_2}$. Sometimes given $\mathcal{K} \subseteq \Pi(D)$, we write simply $\mathcal{K}$ to mean the mapping $I_\mathcal{K}$.

Given $I \subseteq \Pi(D)$, we can introduce set S_I such that $S_I = \{p'\overline{d} : p\overline{d} \in I\}$. We define a function $V_I : D_o \to \mathcal{B}$ as follows: for each $d \in D_o$, $V_I d = \mathbf{T}$ if $d \in S_I^U$, $\mathbf{F}$ otherwise. And V_I is obviously a semivaluation of D. And for all $d \in D_o$, $d \in S_I^U$ only if there is an S_I-derivation for d. This follows from Theorem 4.1.

Theorem 4.2 *Let $I \subseteq \Pi(D)$. $d \in S_I^U$ only if there is a finite $I' \subseteq I$ such that $d \in S_{I'}^U$.*

Proof Assume $d \in S_I^U$. Then by the fact that a derivation sequence is finite, it is clear that there is a finite $I' \subseteq I$ such that there is a finite elementary $S_{I'}$-derivation sequence. □

Definition Let $I \subseteq \Pi(D)$. Then I induces the set $\mathcal{D}^I$ of V_I-complexes based on D and that one-one onto function $\kappa_I : D \to \mathcal{D}^I$ given by the definition of V_I-complexes, and the following functions whose domain is D: the function $\boldsymbol{v}_I$ such that for each $d \in D$, $\boldsymbol{v}_I(d) = (\kappa_I d)^2$, and the function $\boldsymbol{e}_I$ such that for $d \in D$, $\boldsymbol{e}_I d = (\kappa_I d)^\odot$. Let $d \in D_{\alpha \to \beta}$. Then for all $d_1 \in D_\alpha$, $\boldsymbol{e}_I(d)d_1 = \boldsymbol{e}_I(dd_1)$. □

Lemma 4.3 *Let $I_1 \subseteq I_2 \subseteq \Pi(D)$. Then (a) $V_{I_1} \subseteq V_{I_2}$. (b) $\boldsymbol{v}_{I_1} \subseteq \boldsymbol{v}_{I_2}$. (c) $\boldsymbol{e}_{I_1} \subseteq \boldsymbol{e}_{I_2}$.* □

Definition Let $\langle D, J\rangle$ be a general pre-interpretation. An *interpretation* $\mathcal{M}$ is a pair $\langle D, I\rangle$ where I is a type consistent mapping in $\Pi \rightarrow \Phi(D)$. We call $\mathcal{M}$ a D-interpretation. An *assignment φ into $\mathcal{M}$* is a type consistent mapping $\varphi : \Delta \rightarrow D$. When F is a formula in $\mathcal{L}$, we write $\mathcal{M} \models F[\varphi]$ to say that $\mathcal{M}$ satisfies F wrt φ. For all goal formulas G, G_1, G_2, for each rigid atom A, (a) When $s_\alpha, t_\alpha \in T(\Sigma)_\alpha$, $\mathcal{M} \models s_\alpha = t_\alpha[\varphi]$ iff $\mathsf{V}_\varphi s_\alpha = \mathsf{V}_\varphi t_\alpha$, $\mathcal{M} \models s_\alpha \equiv t_\alpha[\varphi]$ iff $\boldsymbol{e}_I(\mathsf{V}_\varphi s_\alpha) = \boldsymbol{e}_I(\mathsf{V}_\varphi t_\alpha)$. (b) $\mathcal{M} \models \top[\varphi]$. (c) $\mathcal{M} \models p(t_1, \cdots, t_n)[\varphi]$ iff $\langle \mathsf{V}_\varphi t_1, \cdots, \mathsf{V}_\varphi t_n\rangle \in Ip$ if p is a constant, or $\langle \mathsf{V}_\varphi t_1, \cdots, \mathsf{V}_\varphi t_n\rangle \in \varphi \circ \boldsymbol{e}_I(p)$ if p is a variable. (d) $\mathcal{M} \models G_1 \vee G_2[\varphi]$ iff $\mathcal{M} \models G_1[\varphi]$ or $\mathcal{M} \models G_2[\varphi]$. (e) $\mathcal{M} \models G_1 \wedge G_2[\varphi]$ iff $\mathcal{M} \models G_1[\varphi]$ and $\mathcal{M} \models G_2[\varphi]$. (f) $\mathcal{M} \models \exists x_\alpha G$ iff there is a $d \in D_\alpha$ such that $\mathcal{M} \models G[\varphi[d/x_\alpha]]$. (g) $\mathcal{M} \models A - G[\varphi]$ iff $\mathcal{M} \models A[\varphi]$ if $\mathcal{M} \models G[\varphi]$.
We write $\mathcal{M} \models F$ to say that a formula F is *valid* in $\mathcal{M}$ if $\mathcal{M} \models F[\varphi]$ for all assignments φ into $\mathcal{M}$. Given a definite program $\mathcal{P}$, we say that $\mathcal{M}$ is a *model* or *D-model* for $\mathcal{P}$, and write $\mathcal{M} \models \mathcal{P}$, if each definite clause in $\mathcal{P}$ is valid in $\mathcal{M}$. Given a closed goal formula G, we say that G is a *logical consequence* of $\mathcal{P}$, and write $\mathcal{P} \models G$ if G is valid in all models of $\mathcal{P}$. □
Definition Let $\mathcal{F} = \langle D, J\rangle$ be a general pre-interpretation, V a semivaluation of D, and $\mathcal{D}$ be the set of V-complexes based on D. Given an $\mathcal{L}$-strucure $\mathcal{A} = \langle \mathcal{D}, J\rangle$ based on $\mathcal{F}$, the *D-interpretation $\mathcal{A}^\odot$ induced by $\mathcal{A}$* is defined to be $\langle D, I\rangle$ where $I = J \circ \kappa \circ (\cdot)^\odot \upharpoonright \Pi$. Conversely, given a D-interpretation $\mathcal{M} = \langle D, I\rangle$ based on $\mathcal{F}$, we can get the set $D^\oplus$ of V_I-complexes based on D. Then $\mathcal{M}^\oplus$ is an *$\mathcal{L}$-structure $\langle D^\oplus, J\rangle$ induced by $\mathcal{M}$.* □

Using the above facts and since assignments into D and $\mathcal{D}$ have one-one correspondence between them, we can show that the two semantics are elemetarily equivalent in the following sense.

Theorem 4.4 *(a) For all formula F in $\mathcal{L}$, $\models F$ iff $\models F$. (b) If $\mathcal{P}$ be a definite program and G a closed goal, then $\mathcal{P} \models G$ iff $\mathcal{P} \models G$.* □

Theorem 4.5 *The extensionality is not valid.*
Proof Take an extensionality formula $p_o \equiv q_o - p_o = q_o$. It is obvious that $\mathcal{V}^2_\varphi p_o = \mathcal{V}^2_\varphi q_o$ does not imply that $\mathcal{V}_\varphi p_o = \mathcal{V}_\varphi q_o$. For the extensionality formula $(\forall x_\alpha \cdot fx \equiv gx) \rightarrow f = g$, we take $\alpha \in \mathcal{T}_0$ and $\beta = o$ and D-interpretation I such that $If = Ig = \mathrm{T}^D_\alpha$. Then $f \equiv g$ but not always $f = g$. □

Let $\mathcal{M} = \langle D, I\rangle$ be an interpretation based on $\mathcal{F} = \langle D, J\rangle$, we can identify $\mathcal{M}$ with the subset I of $\Pi(D)$. And every subset I of $\Pi(D)$ is a D-interpretation. Obviously the set of all D-interpretation is a complete lattice with the usual set inclusion ordering between D-interpretations.

Theorem 4.6 *Let $I_1 \subseteq I_2 \subseteq \Pi(D)$. If $I_1 \models G[\varphi]$, then $I_2 \models G[\varphi]$.*
Proof By induction on G. When G is $\top$, it is obvious. When G is a rigid atom $p(t_1, \cdots, t_n)$, since $I_1 p \subseteq I_2 p$, $I_2 \models G[\varphi]$. When G is $p(t_1, \cdots, t_n)$ where p is a variable. Since $\boldsymbol{e}_{I_1} \subseteq \boldsymbol{e}_{I_2}$, $I_2 \models p(t_1, \cdots, t_n)[\varphi]$. When G is $G_1 \wedge G_2$. $I_1 \models G_1[\varphi]$ and $I_1 \models G_2[\varphi]$. By IH $I_2 \models G_1[\varphi]$ and $I_2 \models G_2[\varphi]$. So $I_2 \models G[\varphi]$. When G is $G_1 \vee G_2$. Assume, wlog, $I_1 \models G_1[\varphi]$. By IH $I_2 \models G_1[\varphi]$. When G is $\exists x_\alpha G_1$. There exists a $d \in D_\alpha$ such that $I_1 \models G_1[\varphi[d/x_\alpha]]$. By IH $I_2 \models G_1[\varphi[d/x_\alpha]]$. So $I_2 \models G[\varphi]$. □

Let $\mathcal{F} = \langle D, J \rangle$ be a general pre-interpretation. We can define a mapping $\mathsf{T}^D_{\mathcal{P}}$ from the lattice of D-interpretations to itself. Let $\mathcal{F}$ be a pre-interpretation $\langle D, J \rangle$ of a definite program $\mathcal{P}$ and I a D-interpretation. Then $\mathsf{T}^D_{\mathcal{P}}(I) = \{p(d_1, \cdots, d_n) \in \Pi(D)$: there exist an assignment φ into D and a clause $p(t_1, \cdots, t_n) \leftarrow G \in \mathcal{P}$ such that $d_i = \mathsf{V}_\varphi t_i$ for each $i \in [n]$ and $I \models G[\varphi]\}$

Lemma 4.7 $\mathsf{T}^D_{\mathcal{P}}$ *is monotonic, i.e. given* $I_1 \subseteq I_2 \subseteq \Pi(D)$, $\mathsf{T}^D_{\mathcal{P}}(I_1) \subseteq \mathsf{T}^D_{\mathcal{P}}(I_2)$.
Proof Assume $p(d_1, \cdots, d_n) \in \mathsf{T}^{\mathcal{F}}_{\mathcal{P}}(I_1)$ for $p(d_1, \cdots, d_n) \in \Pi(D)$. Then there are an assignment φ into I_1 and a clause $p(t_1, \cdots, t_n) \leftarrow G \in \mathcal{P}$ such that $\mathsf{V}_\varphi t_i = d_i$ for all $i \in [n]$ and $I_1 \models G[\varphi]$. By Theorem 4.6, $I_2 \models G[\varphi]$. □

So $\mathsf{T}^{\mathcal{F}}_{\mathcal{P}}$ is a monotonic transformation on the set of all D-interpretations.

Lemma 4.8 *Let* $I \subseteq \Pi(D)$. *Then* $I \models \mathcal{P}$ *iff* $\mathsf{T}^D_{\mathcal{P}}(I) \subseteq I$.
Proof $\Rightarrow$) Assume $p(d_1, \cdots, d_n) \in \mathsf{T}_{\mathcal{P}}(I)$ for some $p(d_1, \cdots, d_n) \in \Pi(D)$. Then there are an assignment φ into D and a clause $p(t_1, \cdots, t_n) - G \in \mathcal{P}$ such that $\mathsf{V}_\varphi t_i = d_i$ for all $i \in [n]$ and $I \models G[\varphi]$. Then since $I \models \mathcal{P}$, $I \models p(t_1, \cdots, t_n)[\varphi]$. Therefore $p(d_1, \cdots, d_n) \in I$.
$\Leftarrow$) Similarly. □

Lemma 4.9 *Let* I_1 *and* I_2 *be D-models of* $\mathcal{P}$. *Then* $I_1 \cap I_2$ *is also D-model of* $\mathcal{P}$.
Proof Since $\mathsf{T}_{\mathcal{P}}(I_1) \subseteq I_1$ and $\mathsf{T}_{\mathcal{P}}(I_2) \subseteq I_2$, by monotonicity of $\mathsf{T}_{\mathcal{P}}$ operator, $\mathsf{T}_{\mathcal{P}}(I_1 \cap I_2) \subseteq \mathsf{T}_{\mathcal{P}}(I_1) \subseteq I_1$ and $\mathsf{T}_{\mathcal{P}}(I_1 \cap I_2) \subseteq \mathsf{T}_{\mathcal{P}}(I_2) \subseteq I_2$. So $\mathsf{T}_{\mathcal{P}}(I_1 \cap I_2) \subseteq I_1 \cap I_2$. □

But the set of all D-models is not closed under join operation, i.e. $I_1 \cup I_2$ is not necessarily a D-model, whenever I_1 and I_2 are D-models. Take for example the definite program $\mathcal{P}_2 = \{p \leftarrow q, r\}$. Then $\Pi(D) = \{p, q, r\}$. $\{q\}$ and $\{r\}$ are D-models for $\mathcal{P}_2$, but $\{q, r\}$ is not a D-model.

Lemma 4.10 *Let* $\langle I_n \rangle_{n \in \omega}$ *be* ω*-chain of D-interpretations. Then for each goal G and assignment* φ *into D,* $\sqcup_{n \in \omega} I_n \models G[\varphi]$ *only if there is an* $n \in \omega$ *such that* $I_n \models G[\varphi]$.
Proof Let $I = \sqcup_{n \in \omega} I_n$. Then $I \models G[\varphi]$ only if $V_I(\mathsf{V}_\varphi G) = \mathbf{T}$. So there is a finite $I' \subseteq I$such that $V_{I'}(\mathsf{V}_\varphi G) = \mathbf{T}$. Therefore there is an $n \in \omega$ such that $I' \subseteq I_n$. By monotonicity $I_n \models G[\varphi]$. □

Lemma 4.11 $\mathsf{T}^D_{\mathcal{P}}$ *is continuous.*
Proof Let $\langle I_n \rangle_{n \in \omega}$ be a ω-chain of D-interpretations. We need to show: $\mathsf{T}_{\mathcal{P}}(\sqcup_{n \in \omega} I_n) = \sqcup_{n \in \omega} \mathsf{T}_{\mathcal{P}}(I_n)$. The monotonicity of $\mathsf{T}_{\mathcal{P}}$ implies that $\sqcup_{n \in \omega} \mathsf{T}_{\mathcal{P}}(I_n) \subseteq \mathsf{T}_{\mathcal{P}}(\sqcup_{n \in \omega} I_n)$. Now we need to show that $\mathsf{T}_{\mathcal{P}}(\sqcup_{n \in \omega} I_n) \subseteq \sqcup_{n \in \omega} \mathsf{T}_{\mathcal{P}}(I_n)$. Let $d_1, \cdots, d_n \in D$, and $p(d_1, \cdots, d_n) \in \Pi(D)$. Assume $p(d_1, \cdots, d_n) \in \mathsf{T}_{\mathcal{P}}(\sqcup_{n \in \omega} I_n)$, to show $p(d_1, \cdots, d_n) \in \sqcup_{n \in \omega} \mathsf{T}_{\mathcal{P}}(I_n)$. There are $p(t_1, \cdots, t_n) \leftarrow G \in \mathcal{P}$ and an assignment φ into H such that $\varphi t_i = d_i$ for all $i \in [n]$ and $\sqcup_{n \in \omega} I_n \models G[\varphi]$. So there is $n \in \omega$ such that $I_n \models G[\varphi]$. Therefore there is $n \in \omega$ such that $p(d_1, \cdots, d_n) \in \mathsf{T}_{\mathcal{P}}(I_n)$. □

So we can show that every definite program has the least D-model as follows:

Theorem 4.12 $(\mathsf{T}^D_{\mathcal{P}})^\omega(\phi)$ *is the least fixpoint of* $\mathsf{T}^D_{\mathcal{P}}$. □

Theorem 4.13 *Let* $\mathsf{M}^D_{\mathcal{P}} = \bigcap\{I \subseteq \Pi(D) : I \models P\}$. *then* $\mathsf{M}^D_{\mathcal{P}}$ *is the least D-model of* $\mathcal{P}$ *and* $\mathsf{M}^D_{\mathcal{P}} = \mathsf{T}^\omega_{\mathcal{P}}(\phi)$.
Proof By Lemmas 4.8,4.9,4.7 and Theorem 4.12. □

5 Herbrand Models

In order to determine validity or logical consequences, we need to consider all interpretations of the language $\mathcal{L}$. In this section we shall show that we can restrict our attention to Herbrand models. That is, we show that if A is true in all Herbrand (that is symbolic) models it follows that A is true in all models and a fortiori in the model intended by the person who wrote the program.

Definition The *Herbrand frame* H is a set such that (a) H is the set of all ρ-normal c-terms. (b) Let $f \in H_{\alpha\to\beta}$, then for all $t \in H_\alpha$, $f(t) = \rho(ft)$. □

It is obvious that the Herbrand frame H is countable.

Definition The *Herbrand pre-interpretation* $\mathcal{HF}$ is a pre-interpretation $\langle H, J\rangle$ such that H is the Herbrand frame and J satisfies the following: (a) If c_α is a constant such that α is a primitive type, then $Jc_\alpha = c_\alpha$. (b) If $d_{\alpha\to\beta}$ is a constant of type $\alpha \to \beta$, then for all $t_\alpha \in H_\alpha$, $(Jd_{\alpha\to\beta})(t_\alpha) = d_{\alpha\to\beta}t_\alpha$. □

Lemma 5.1 *The Herbrand pre-interpretation is general.* □

Definition An *Herbrand interpretation* $\mathcal{M}$ is an interpretation $\langle H, I\rangle$ based on the Herbrand pre-interpretation. The *Herbrand base* $\mathcal{HB}$ is the set $\Pi(H)$. □

Let $I \subseteq \Pi(H)$ be an Herbrand interpretation and φ an assignment into I. Then we can consider φ as the generalized substitution σ such that for each term $t \in T(\Sigma)$, $\sigma t = (\varphi \uparrow FV(t))t$. It is easy to see that for every term t, φt is a c-term and $\mathsf{V}_\varphi t = \varphi t$, for each goal formula G, φG a closed goal formula, and for each definite clause C, φC a closed definite clause.

Let I be a D-interpretation based on $\mathcal{F}$. The *Herbrand interpretation* I^* *induced by* I is an Herbrand interpretation such that for every $A \in \Pi(H)$, $A \in I^*$ iff $I \models A$. Let φ and φ' be assignments into H and D, respectively. Then we say that φ' is *induced by* φ if $\varphi' = \varphi \circ \mathsf{V}^{\mathcal{F}}$. The mapping $\mathsf{V}^{\mathcal{F}} : H \to D$ is a *homomorhism from* I^* *into* I, since for $p \in \Pi_{\alpha_1,\cdots,\alpha_n\to o}, h_i \in H_{\alpha_i}, i \in [n]$, if $\langle h_1, \cdots, h_n\rangle \in I^*p$, then $\langle \mathsf{V}^{\mathcal{F}}h_1, \cdots, \mathsf{V}^{\mathcal{F}}h_n\rangle \in Ip$. Let $h \in H_{\alpha_1,\cdots,\alpha_n\to o}$. Then for all $h_i \in H_{\alpha_i}, i \in [n]$, $\langle h_1, \cdots, h_n\rangle \in \boldsymbol{e}_{I^*}(h)$ implies $\langle \mathsf{V}^{\mathcal{F}}h_1, \cdots, \mathsf{V}^{\mathcal{F}}h_n\rangle \in \boldsymbol{e}_I(\mathsf{V}^F h)$.

Lemma 5.2 *Let $I, I^*, \varphi', \varphi$ be as above. Then (a) If t is a term, then $\mathsf{V}^{\mathcal{F}}_{\varphi'}(\varphi t) = \mathsf{V}^{\mathcal{F}}_{\varphi'}t$, (b) If A is a rigid atom then $I^* \models A[\varphi]$ iff $I \models A[\varphi']$, (c) If G is a goal formula such that $I^* \models G[\varphi]$, then $I \models G[\varphi']$, (d) If C is a definite clause such that $I \models C[\varphi']$, then $I^* \models C[\varphi]$, (e) Then if $I \models \mathcal{P}$, then $I^* \models \mathcal{P}$.* □

Let $\mathcal{F}$ be a general pre-interpretation. Then $\models_{\mathcal{F}}$ denotes logical implication in the context of fixed domains and functional assignment. Specifically $\models_{\mathcal{HF}}$ denotes logical implication in the context of Herbrand frame and functional assignment.

Let G be a goal formula. We write $\exists(G)$ to denote the existential closure of free variables in G.

Theorem 5.3 *Let $\mathcal{P}$ be a definite program and G a goal formula. Then $\mathcal{P} \models \exists(G)$ iff $\mathcal{P} \models_{\mathcal{HF}} \exists(G)$.*
Proof $\Longleftarrow$) Let an Herbrand interpretation induced by the given interpretation I be I^*. Assume $I \models \mathcal{P}$. Then $I^* \models \mathcal{P}$, so $I^* \models \exists(G)$. Then there is an assignment φ into I^* such that $I^* \models G[\varphi]$. Let the assignment φ' into I be induced by φ. Then $I \models G[\varphi']$ by Lemma 5.2 (c). So $I \models \exists(G)$.□

If φ is a substitution, then φ_{-x_α} is that substitution σ such that $\sigma = \varphi \uparrow (\Delta - \{x_\alpha\})$.

Lemma 5.4 *Let $I \subseteq \Pi(H)$. Then for all closed substitution σ, assignment φ into H, and goal formula G, $I \models \sigma G[\varphi]$ iff $I \models \varphi\sigma G$.*
Proof We prove by induction on G. When G is $\top$ or a rigid atom, it is obvious. When G is $p(t_1, \cdots, t_n)$ where $p \in \Delta$. $I \models \sigma G[\varphi]$ iff $\langle \varphi\sigma t_1, \cdots, \varphi\sigma t_n\rangle \in e_I(\varphi\sigma p)$ iff $\langle \varphi'\varphi\sigma t_1, \cdots, \varphi'\varphi\sigma t_n\rangle \in e_I(\varphi'[\varphi\sigma p/p]p)$ for all assignment φ' into H iff $I \models \varphi\sigma G[\varphi']$ for all assignment φ' into H iff $I \models \varphi\sigma G$.

When G is $\exists x_\alpha G_1$. $I \models \sigma G[\varphi]$ iff $I \models \exists x_\alpha \sigma_{-x_\alpha} G_1[\varphi]$ iff there is an $h \in H_\alpha$ such that $I \models \sigma_{-x_\alpha} G_1[\varphi[h/x_\alpha]]$ iff there is an $h \in H_\alpha$ such that $I \models \varphi[h/x_\alpha]\sigma_{-x_\alpha} G_1$ by IH iff for all assignment φ' into H, $I \models \varphi'\varphi[h/x_\alpha]\sigma_{-x_\alpha} G_1$, since $\varphi[h/x_\alpha]\sigma_{-x_\alpha} G_1$ is a closed goal. iff $I \models (\varphi'[h/x_\alpha])\varphi_{-x_\alpha}\sigma_{-x_\alpha} G_1$ iff $I \models \varphi_{-x_\alpha}\sigma_{-x_\alpha} G_1[\varphi'[h/x_\alpha]]$ iff $I \models \exists x_\alpha \varphi_{-x_\alpha}\sigma_{-x_\alpha} G_1[\varphi']$ iff $I \models \varphi\sigma G$. □

Corollary 5.5 *For all assignment φ into H, goal formula G, $I \models G[\varphi]$ iff $I \models \varphi G$.* □

Theorem 5.6 *For all closed substitution σ and goal formula G such that $\sigma\exists x_\alpha G$ is closed, $I \models \sigma\exists x_\alpha G$ iff there is an $h \in H_\alpha$ such that $I \models \sigma[h/x_\alpha]G$.*
Proof Let φ be an assignment into H. $I \models \sigma\exists x_\alpha G[\varphi]$ iff $I \models \exists x_\alpha \sigma_{-x_\alpha} G[\varphi]$ iff there is an $h \in H_\alpha$ such that $I \models \sigma_{-x_\alpha} G[\varphi[h/x_\alpha]]$ iff there is an $h \in H_\alpha$ such that $I \models \varphi[h/x_\alpha]\sigma_{-x_\alpha} G$ by Corollary 5.5 iff $I \models \sigma[h/x_\alpha]G[\varphi]$ by Corollary 5.5, since $\varphi[h/x_\alpha]\sigma_{-x_\alpha} = \varphi\sigma[h/x_\alpha]$. □

Corollary 5.7 *Let $\mathsf{M}^H_\mathcal{P} = \bigcap\{I \subseteq \Pi(H) : I \models \mathcal{P}\}$. Then $\mathsf{M}^H_\mathcal{P} \models \mathcal{P}$.*
Proof Follows from Theorem 4.13. □

Theorem 5.8 *$(\mathsf{T}^H_\mathcal{P})^\omega(\phi)$ is the least fixed point of $\mathsf{T}^H_\mathcal{P}$ and $\mathsf{M}^H_\mathcal{P} = (\mathsf{T}^H_\mathcal{P})^\omega(\phi)$.*
Proof Follows from Lemma 4.11. □

Theorem 5.9 *Let $A \in \Pi(H)$. Then $\mathcal{P} \models A$ iff $\mathsf{M}^H_\mathcal{P} \models A$.*
Proof $\mathcal{P} \models A$ iff $\mathcal{P} \models_{\mathcal{HF}} A$ iff for all H-interpretation I such that $I \models \mathcal{P}$, $A \in I$ iff $A \in \mathsf{M}^H_\mathcal{P}$. □

For the definite program $\mathcal{P}_1$ introduced in section 1, it is easy to see that

$$\mathsf{T}^\omega_{\mathcal{P}_1}(\phi) = \{p(a), q(a), r(p(a))\}$$

So $r(p(a))$ is a logical consequence of $\mathcal{P}_1$, while $r(q(a))$ is not.

The program $\mathcal{P}_1$ is non-extensional in the sense that extensional identity of arguments of the predicate r does not imply extensional identity of proposition $r(\cdot)$. In [30] Wadge defined a fragment of higher-order logic programming language (in fact it's a pure subset of HiLog [8]) where every program behaves extensionally.

Example We can define the following higher-order logic program $\mathcal{P}_3$ in the language of [30]: Let MAP be predicate constant of type $(int \to o), list \to o$ and $\cdot$ be an infix functional constant of type $int, list \to list$ and p and q predicate constants of type $int \to o$ and $\mathcal{P}_3$ include the following definite clauses.
$MAP(z, x \cdot l) \leftarrow zx \wedge MAP(z, l)$.
$MAP(z, nil) \leftarrow \top$.

Assume that the above clauses are the only clauses that defines the predicate MAP. Let I be a fixpoint of $\mathsf{T}_{\mathcal{P}_3}$. We shall show that $p \equiv q \to MAPp \equiv MAPq$ is valid under I. Let $p \equiv q$ valid under I. Then for all $a \in H_{int}$, $pa \in I$ iff $qa \in I$. Moreover the set H_{list} has the following inductive characteriztion. (a) $nil \in H_{list}$. (b) For $a \in H_{int}$, $a \cdot l \in H_{list}$ if $l \in H_{list}$. To prove $MAPp \equiv MAPq$ is valid in I, it's enough to show that for all $l \in H_{list}$, $MAP(p,l) \in I$ iff $MAP(q,l) \in I$. We prove this by induction on l. Obviously $MAP(p, nil), MAP(q, nil) \in I$. Let $a \cdot l \in H_{list}$. Assume $MAP(p, a \cdot l) \in I$ to show $MAP(q, a \cdot l) \in I$. Then $pa, MAP(p,l) \in I$. So by IH, $MAP(q,l) \in I$. Therefore $MAP(q, a \cdot l) \in I$. □

6 Completeness

In this section we prove completeness of interpreter in [26]. Our actual interpreter is that of [26] plus backchaining when atomic goals need to be solved. The definition of this non-deterministic interpreter can be given by describing how a theorem prover for programs and goals should function. This interpreter, given the pair $\langle \mathcal{P}, G \rangle$ in its initial state, should either succeed or fail. We shall use the notation $\mathcal{P} \vdash G$ to indicate the meta proposition that the interpreter succeeds if started in the state $\langle \mathcal{P}, G \rangle$. The search related semantics which we want to attribute to the logical constants can be specified as follows: (a) $\mathcal{P} \vdash \top$. (b) $\mathcal{P} \vdash G_1 \vee G_2$ only if $\mathcal{P} \vdash G_1$ or $\mathcal{P} \vdash G_2$. (c) $\mathcal{P} \vdash G_1 \wedge G_2$ only if $\mathcal{P} \vdash G_1$ and $\mathcal{P} \vdash G_2$. (d) $\mathcal{P} \vdash \exists x_\alpha G_1$ only if there is some term $t \in T(\Sigma)_\alpha$ such that $\mathcal{P} \vdash [t/x_\alpha]G_1$. (e) $\mathcal{P} \vdash A$ only if there are a definite clause $A_1 \leftarrow G_1 \in \mathcal{P}$ and a substitution σ such that $A = \sigma A_1$ and $\mathcal{P} \vdash \sigma G_1$.

Let F be a formula of $\mathcal{L}$. Then $|F|$ denotes the set $\{\varphi F : \varphi$ is an assignment into $H\}$. It is easy to see that if F is a goal formula, $|F|$ is a set of closed goal formulas, and if F is a definite clause, then $|F|$ is a set of closed definite clauses. This notation can be extended to set Γ of formulas of $\mathcal{L}$: $|\Gamma| = \bigcup\{|F| : F \in \Gamma\}$.

Definition Let Γ be a set of formulas that are either closed atoms or definite clauses, and let G be a closed goal formula. Then a Γ-*derivation sequence for* G is a finite sequence $G^1, G^2, \cdots, G^n$ of closed goal formulas such that G^n is G, and for each $i \in [n]$, (a) if G^i is a closed atom, then i) G^i is $\top$, or ii) $G^i \in \Gamma$, or iii) there is a definite clause $G^i \leftarrow G^j \in |\Gamma|$ such that $j < i$, (b) if G^i is $G_1 \vee G_2$, then for some $j < i$, G^j is either G_1 or G_2, (c) if G^i is $G_1 \wedge G_2$, then for some $j, k < i$, $G^j = G_1$ and $G^k = G_2$, (d) if G^i is $\exists x_\alpha G_1$, then there is a $t \in H_\alpha$ and $j < i$ such that $[t/x_\alpha]G_1 = G^j$. □

Theorem 6.1 *Let $I \subseteq \Pi(H)$. Then for all closed goal formula G, $I \models G$ iff there is an I-derivation sequence for G.*

Proof $\Leftarrow$) Let $G^1, \cdots, G^n$ be an I-derivation sequence. We prove by induction on i: for all $i \in [n]$, $I \models G^i$. When $i = 1$, then it is obvious. When $i > 1$. If $G^i = G_1 \wedge G_2$, then by IH, $I \models G_1$ and $I \models G_2$. So $I \models G_1 \wedge G_2$. If $G^i = \exists x_\alpha G_1$, then by IH, there is a $t \in H_\alpha$ such that $I \models [t/x_\alpha]G_1$. So $I \models \exists x_\alpha G_1$ by Theorem 5.6.

$\Rightarrow$) Follows from Theorem 4.1, since for a Herbrand interpretation I, we can identify I with S_I. □

Lemma 6.2 *Let G be a closed goal formula. Then $\mathcal{P} \vdash G$ iff there is a $\mathcal{P}$-derivation for G.*

Proof See [28]. □

Theorem 6.3 *Let G be a closed goal formula. Then $\mathcal{P} \vdash G$ iff $\mathcal{P} \models G$.*
Proof By Theorems 5.9,5.8, $\mathcal{P} \models G$ iff $\mathsf{T}^{\omega}_{\mathcal{P}}(\phi) \models G$. Let $I_n = \mathsf{T}^{n}_{\mathcal{P}}(\phi)$ for $n \in \omega$. Now we need to prove that there is a $\mathcal{P}$-derivation $G^1, \cdots, G^l$ for G iff there is an $n \in \omega$ such that $I_n \models G$.
$\Rightarrow$) By induction on l. When G is $\top$, $I_0 \models \top$. When G is $G_1 \wedge G_2$, then there are $\mathcal{P}$-derivations for G_1 and G_2 whose lengths are less than l. So by IH, there are $n_1, n_2 \in \omega$ such that $I_{n_1} \models G_1$ and $I_{n_2} \models G_2$. Assume, wlog, $n_1 < n_2$. Then $I_{n_2} \models G_2$, so $I_{n_2} \models G_1 \wedge G_2$. When G is $\exists x_\alpha G_1$. Then there are a term $t \in H_\alpha$ and a $\mathcal{P}$-derivation for $[t/x_\alpha]G_1$ whose length is less than l. So by IH, there is an $n \in \omega$ such that $I_n \models [t/x_\alpha]G_1$. Therefore $I_n \models \exists x_\alpha G_1$ by Theorem 5.6. When G is a rigid atom A. Then there are a number $j < l$ and a definite clause $A \leftarrow G^j \in |\mathcal{P}|$. By IH, $I_n \models G^j$. Therefore $I_{n+1} \models A$.
$\Leftarrow$) We prove the claim by induction on n. First assume the claim true if $I_n \models G$. To prove the claim for $n+1$ assume $I_{n+1} \models G$. Then there is an I_{n+1}-derivation $G^1, \cdots, G^m$ for G by Theorem 6.1. Now we prove, by induction on i, that there is a $\mathcal{P}$-derivation for G^i, for each $i \in [m]$. If G^i is $\top$, it is immediate. If G^i is a rigid atom A, then since $A \in I_{n+1}$, there is a definite clause $A \leftarrow G_1 \in |\mathcal{P}|$ such that $I_n \models G_1$. Then by our first assumption, there is a $\mathcal{P}$-derivation for G_1. We now get a $\mathcal{P}$ derivation for A by appending A to this sequence. When G^i is $G_1 \wedge G_2$. Then by our second IH, there are $\mathcal{P}$-derivations for G_1 and G_2. Now we get a $\mathcal{P}$-derivation for G^i by appending G^i to the end of concatenation these sequences. When G^i is $\exists x_\alpha G_1$. By second IH, there is a term $t \in H_\alpha$ such that there is a $\mathcal{P}$-derivation for $[t/x_\alpha]G_1$, to which we attach G^i to get $\mathcal{P}$-derivation for G^i. □

7 Equality and $D/\mathcal{E}$-Interpretations

Much of the research in logic programming concentrates on extensions of Prolog. An important issue is the integration of the essential concepts of functional and logic programming. Another issue is the use of equations to define data types. Works along these lines can be found in [11, 18].

In this section we will develop semantics for higher-order logic programs augmented with an equality theory $\mathcal{E}$. We will establish the existence of the least model and least fixpoint semantics.

Let D be a frame and R an equivalent relation on D. For each type $\alpha \in \mathcal{T}$, we write R_α for the restriction of R to D_α. Then

$$\mathsf{R} = \bigcup_{\alpha \in \mathcal{T}} \mathsf{R}_\alpha.$$

Let d be an element of D_α. Then $[d]_\mathsf{R}$ is the equivalent class containing d. We also say that R is a *congruence relation* on D if R is an equivalent relation on D and for all $\alpha, \beta \in \mathcal{T}$, for all $d \in D_{\alpha\rightarrow\beta}$, for all $c \in D_\alpha$, $[d]_\mathsf{R}[c]_\mathsf{R} = [dc]_\mathsf{R}$. We sometimes write $[d]_\mathsf{R}$ as $[d]$ when the congruence relation is clear from the context.

Since $=$ is a binary predicate symbol, any interpretation of it should be a binary relation R over a frame D. And R should be an equivalence relation and congruence relation, because R must satisfy the following axioms of equality.

$$x = x$$
$$x = y - y = x$$
$$x = y \wedge y = z - x = z$$
$$f = g \wedge x = y - fx = gy$$

Given a frame D and a congruence relation R on D, we define a quotient frame D/R as a frame $\{D_\alpha/\mathsf{R}_\alpha\}_\alpha$.

If $\mathcal{F} = \langle D, J\rangle$ is a pre-interpretation, $\mathcal{F}/\mathsf{R}$ is defined to be a pre-interpretation $\langle D/\mathsf{R}, J'\rangle$ such that for each constant c, $J'c = [Jc]_\mathsf{R}$.

Lemma 7.1 *Let* R *be a congruence relation on* D*. If a pre-interpretation* $\mathcal{F} = \langle D, J\rangle$ *is general, then* $\mathcal{F}/\mathsf{R}$ *is also general.*

Proof Let V be a valuation function in $\mathcal{F}$, and $\mathcal{F}/\mathsf{R} = \langle D/\mathsf{R}, J'\rangle$ and φ an assignment into $\mathcal{F}/\mathsf{R}$. Then there is an assignment φ' into $\mathcal{F}$ such that $\varphi = \varphi' \circ [\cdot]_\mathsf{R}$. Define a binary function V' such that for each term t, $\mathsf{V}'_\varphi t = [\mathsf{V}_{\varphi'} t]_\mathsf{R}$. We show V' is a valuation function in $\mathcal{F}/\mathsf{R}$ by showing that for each term t_α, $\mathsf{V}'_\varphi t_\alpha \in D_\alpha/\mathsf{R}_\alpha$ by induction on t_α.
When t_α is a variable x_α, $\mathsf{V}'_\varphi x_\alpha = [\mathsf{V}_{\varphi'} x_\alpha] = [\varphi' x_\alpha] = \varphi x_\alpha$. When t_α is a constant c_α, $\mathsf{V}'_\varphi c_\alpha = [\mathsf{V}_{\varphi'} c_\alpha] = [Jc_\alpha] = J'c_\alpha$ When t_α is $f_{\beta\to\alpha} s_\beta$,

$$\begin{aligned} \mathsf{V}'_\varphi(f_{\beta\to\alpha} s_\beta) &= [\mathsf{V}_{\varphi'}(f_{\beta\to\alpha} s_\beta)] \\ &= [(\mathsf{V}_{\varphi'} f_{\beta\to\alpha})(\mathsf{V}_{\varphi'} s_\beta)] \\ &= [\mathsf{V}_{\varphi'} f_{\beta\to\alpha}][\mathsf{V}_{\varphi'} s_\beta] && \text{by definition of R} \\ &= (\mathsf{V}'_\varphi f_{\beta\to\alpha})(\mathsf{V}'_\varphi s_\beta) && \text{by induction hypothesis} \end{aligned}$$

When t_α is $\lambda x_\beta s_\gamma$, let

$$\begin{aligned} d' = \mathsf{V}'_\varphi(\lambda x_\beta s_\gamma) &= [\mathsf{V}_{\varphi'} \lambda x_\beta s_\gamma] \\ &= [\lambda b \in D_\beta \cdot \mathsf{V}_{\varphi'[b/x_\beta]} s_\gamma] \end{aligned}$$

For $b \in D_\beta$,

$$\begin{aligned} d'[b] &= [\mathsf{V}_{\varphi'} \lambda x_\beta s_\gamma][b] \\ &= [(\mathsf{V}_{\varphi'} \lambda x_\beta s_\gamma) b] && \text{by definition of R} \\ &= [\mathsf{V}_{\varphi'[b/x_\beta]} s_\gamma] \\ &= \mathsf{V}'_{\varphi[[b]/x_\beta]} s_\gamma && \text{by induction hypothesis} \end{aligned}$$

□

Corollary 7.2 *Let* $\mathcal{HF}$ *be the Herbrand pre-interpretation, then* $\mathcal{HF}/\mathsf{R}$ *is general.* □

In the remaining of this section we assume that every pre-interpretation we mention is general. Since a congruence relation R over a pre-interpretation $\mathcal{F}$ can be taken as an interpretation of the equality symbol $=$, we have

Proposition 7.3 *Let* $\mathcal{P}$ *be a program,* $\mathcal{E}$ *an equality theory and* A *be a closed atom. Then*

$$\mathcal{P}, \mathcal{E} \models A \iff \mathcal{P}, \mathcal{E} \models_{\mathcal{F}/\mathsf{R}} A \text{ for all pre-interpretation } \mathcal{F} \text{ and congruence relation } \mathsf{R} \text{ over } \mathcal{F}$$

□

We want the existence of a canonical model for the equality theory, i.e. we wish the existence of a congruence R_0 over $\mathcal{HF}$ such that

Proposition 7.4 *$\mathcal{E} \models s_\alpha = t_\alpha$ iff $[s_\alpha]_{\mathsf{R}_0} = [t_\alpha]_{\mathsf{R}_0}$ where s_α and t_α are closed terms.* □

But this can be achieved only if the theory $\mathcal{E}$ has a finest congruence relation R_0. This motivates our choice of using *Horn equality clauses* in our framework presented below.

A *definite clause logic program* $\mathcal{P}$ is defined to be a finite set of definite clauses

$$A \leftarrow e_1 \wedge \cdots \wedge e_n \wedge G$$

where A is a rigid atom in $\mathcal{G}oal$, i.e. not an equation, each e_i is an equation and G is a goal formula in $\mathcal{G}oal$.

A *Horn equality clause* takes one of two forms

$$e \leftarrow e_1 \wedge \cdots \wedge e_n$$

or

$$\leftarrow e_1 \wedge \cdots \wedge e_n$$

where $n \geq 0$ and all the e_i's therein are equations. As usual, variables in Horn equality clauses are implicitly universally quantified. We define a *Horn clause equality theory* to be a set of equality clauses. A given consistent Horn clause equality theory $\mathcal{E}$ defines a logic programming language whose programs, called *definite logic programs*, are the pairs $(\mathcal{P}, \mathcal{E})$ where $\mathcal{P}$ is a definite clause logic program.

Lemma 7.5 *Let $\mathcal{F}$ be a pre-interpretation $\langle D, J \rangle$. Then there exists a finest congruence over $\mathcal{F}$ generated by each consistent Horn clause equality theory $\mathcal{E}$.*

Proof Consider models of $\mathcal{E}$ over the frame D, and for our purposes here, a model is a set of pairs from D. Suppose now that I is the intersection of a set of models of $\mathcal{E}$. If I is not a model itself, then there are a clause C of the form

$$s = t \leftarrow s_1 = t_1 \wedge \cdots \wedge s_n = t_n$$

or of the form

$$\leftarrow s_1 = t_1 \wedge \cdots \wedge s_n = t_n$$

and an assignment φ into D such that I does not satisfy C under φ. Let

$$\begin{array}{ll} \mathsf{V}_\varphi s = c, & \mathsf{V}_\varphi t = d \\ \mathsf{V}_\varphi s_i = c_i & \mathsf{V}_\varphi t_i = d_i, \quad i \in [n]. \end{array}$$

Then if C is of the first form then $\langle c_i, d_i \rangle \in I$ for all $i \in [n]$, while $\langle c, d \rangle \notin I$, contradicting the fact that $\langle c, d \rangle$ is in the models of the set in question. If C is of the second form then $\langle c_i, d_i \rangle \in I$ for all $i \in [n]$, which is clearly impossible. The finest congruence then is given by the intersection of all models of $\mathcal{E}$. □

We thus may now write $\mathcal{F}/\mathcal{E}$ to denote this finest congruence. In a situation where both $\mathcal{F}$ and $\mathcal{F}/\mathcal{E}$ are being discussed, we write V for the evaluation function in $\mathcal{F}$ and write V' for the evaluation function in $\mathcal{F}/\mathcal{E}$.

Corollary 7.6 *Let C be a clause of $\mathcal{E}$ then C is valid under $\mathcal{F}/\mathcal{E}$.* □

As a consequence

Lemma 7.7 *Let $(\mathcal{P},\mathcal{E})$ be a definite program and A a closed atom. Then*

$$(\mathcal{P},\mathcal{E}) \models A \iff (\mathcal{P},\mathcal{E}) \models_{\mathcal{F}/\mathcal{E}} A \quad \textit{for all pre-interpretation } \mathcal{F}.$$

Proof Let $\mathcal{F} = \langle D, J\rangle$. Then it suffices to prove that $(\mathcal{P},\mathcal{E}) \models_{\mathcal{F}/\mathcal{E}} A \iff (\mathcal{P},\mathcal{E}) \models_{\mathcal{F}/\mathsf{R}} A$ for all R. $\Longrightarrow$) Let R_0 be the finest congruence relation. For some R, let I be any D/R-interpretation such that $I \models (\mathcal{P},\mathcal{E})$, but $I \not\models A$. Construct the following $D/\mathcal{E}$-model I' by defining that $I' \models p([d_1]_{\mathsf{R}_0},\cdots,[d_n]_{\mathsf{R}_0})$ iff $I \models p([d_1]_{\mathsf{R}},\cdots,[d_n]_{\mathsf{R}})$ for all predicate constant p. This is well defined because R_0 is finer than R. It is now easy to see that $I' \models (\mathcal{P},\mathcal{E})$ but $I' \not\models A$. □

Lemma 7.8 *Let $(\mathcal{P},\mathcal{E})$ be a definite program and A a closed atom. Then*

$$(\mathcal{P},\mathcal{E}) \models A \iff \mathcal{P} \models_{\mathcal{F}/\mathcal{E}} A \quad \textit{for all pre-interpretation } \mathcal{F}.$$

Proof This lemma follows from Lemma 7.7 and Corollary 7.6. □

Theorem 7.9 *Let $(\mathcal{P},\mathcal{E})$ be a definite program and A a closed atom. Then*

$$(\mathcal{P},\mathcal{E}) \models A \iff \mathcal{P} \models_{\mathcal{HF}/\mathcal{E}} A.$$

Proof This theorem follows from Lemma 7.8 and the fact that $\mathcal{P}$ is in clausal form. □

We now give definitions with respect to a given logic program $(\mathcal{P},\mathcal{E})$.

We consider the fixpoint formalization of an intuitive semantics of our logic programs. Let $\mathcal{F}$ be a pre-interpretation $\langle D, J\rangle$. Then $\mathsf{T}_{(\mathcal{P},\mathcal{E})}$ maps from and into $D/\mathcal{E}$-interpretations and is defined as follows: for $D/\mathcal{E}$-interpretation I,

$$\begin{aligned}\mathsf{T}_{(\mathcal{P},\mathcal{E})}(I) = \{p(a_1,\cdots,a_n) \;&\in \Pi(D/\mathcal{E}) : \text{there are a clause}\\ &p(t_1,\cdots,t_n) \leftarrow e_1 \wedge\cdots\wedge e_m \wedge G \text{ in } \mathcal{P}\\ &\text{and an assignment into } D/\mathcal{E} \text{ such that}\\ &\mathsf{V}'_\varphi t_i = a_i, \text{ for } i \in [n] \text{ and}\\ &I \models e_1 \wedge\cdots\wedge e_m \wedge G[\varphi]\}.\end{aligned}$$

Lemma 7.10 $\mathsf{T}_{(\mathcal{P},\mathcal{E})}$ *is monotonic.*
Proof Let $I_1 \subseteq I_2 \subseteq \Pi(D/\mathcal{E})$. Assume $p(a_1,\cdots,a_n) \in \mathsf{T}_{(\mathcal{P},\mathcal{E})}(I_1)$. Then there are a clause $p(t_1,\cdots,t_n) \leftarrow e_1 \wedge\cdots\wedge e_m \wedge G \in \mathcal{P}$ and an assignment φ into $D/\mathcal{E}$ such that $\mathsf{V}'_\varphi t_i = a_i$ for $i \in [n]$ and $I_1 \models e_1 \wedge\cdots\wedge e_m \wedge G[\varphi]$. So $I_1 \models G[\varphi]$ and for all $j \in [m]$, $I_1 \models e_j[\varphi]$. Since satisfaction of equations does not depend on interpretations, $I_2 \models e_j[\varphi]$, for all $j \in [m]$. And by Theorem 4.6 $I_2 \models G[\varphi]$. Therefore $I_2 \models e_1 \wedge\cdots\wedge e_m \wedge G[\varphi]$. □

We can prove following lemma similarly as Lemma 4.8.

Lemma 7.11 *Let $I \subseteq \Pi(D/\mathcal{E})$. Then*

$$I \models \mathcal{P} \Longleftrightarrow \mathsf{T}_{(\mathcal{P},\mathcal{E})}(I) \subseteq I.$$

□

Using above lemma and monotonicity of $\mathsf{T}_{(\mathcal{P},\mathcal{E})}$ we have the intersection properties of models.

Lemma 7.12 *Let I_1 and I_2 be $D/\mathcal{E}$-models of $(\mathcal{P},\mathcal{E})$.*
Then $I_1 \cap I_2$ also a $D/\mathcal{E}$-model of $(\mathcal{P},\mathcal{E})$. □

We can now establish the existence of the least model.

Theorem 7.13 *There is the least $D/\mathcal{E}$-model of $(\mathcal{P},\mathcal{E})$.*
Proof By above lemma the intersection of all $D/\mathcal{E}$-models of $(\mathcal{P},\mathcal{E})$ is itself a $D/\mathcal{E}$-model of $(\mathcal{P},\mathcal{E})$, which is obviously the least $D/\mathcal{E}$-model. □

Lemma 7.14 $\mathsf{T}_{(\mathcal{P},\mathcal{E})}$ *is continuous.*
Proof Let $\langle I_k\rangle_k$ be an ω-chain of $D/\mathcal{E}$-interpretations and $I_\omega = \sqcup_k I_k$. We now need to show that

$$\mathsf{T}_{(\mathcal{P},\mathcal{E})}(I_\omega) = \sqcup_k \mathsf{T}_{(\mathcal{P},\mathcal{E})}(I_k).$$

By monotonicity of $\mathsf{T}_{(\mathcal{P},\mathcal{E})}$ we have

$$\sqcup_k \mathsf{T}_{(\mathcal{P},\mathcal{E})}(I_k) \subseteq \mathsf{T}_{(\mathcal{P},\mathcal{E})}(I_\omega).$$

In order to establish

$$\mathsf{T}_{(\mathcal{P},\mathcal{E})}(I_\omega) \subseteq \sqcup_k \mathsf{T}_{(\mathcal{P},\mathcal{E})}(I_k),$$

assume $p(a_1,\cdots,a_n) \in \mathsf{T}_{(\mathcal{P},\mathcal{E})}(I_\omega)$. Then there are a clause $p(t_1,\cdots,t_n) \leftarrow e_1 \wedge \cdots \wedge e_m \wedge G$ in $\mathcal{P}$ and an assignment φ into $D/\mathcal{E}$ such that $\mathsf{V}'_\varphi t_i = a_i$ for $i \in [n]$ and $I_\omega \models e_1 \wedge \cdots \wedge e_m \wedge G[\varphi]$. Then $I_\omega \models G[\varphi]$, so there is a $k \in \omega$ such that $I_k \models G[\varphi]$. And as before $I_k \models e_j[\varphi]$ for $j \in [m]$. So $I_k \models e_1 \wedge \cdots \wedge e_m \wedge G[\varphi]$. Therefore there is a $k \in \omega$ such that $p(a_1,\cdots,a_n) \in \mathsf{T}_{(\mathcal{P},\mathcal{E})}(I_k)$. □

By continuity of $\mathsf{T}_{(\mathcal{P},\mathcal{E})}$, and Lemma 7.11, we now have

Theorem 7.15 $\mathsf{T}^\omega_{(\mathcal{P},\mathcal{E})}(\emptyset)$ *is the least fixpoint of $\mathsf{T}_{(\mathcal{P},\mathcal{E})}$ and the least $D/\mathcal{E}$-model of $(\mathcal{P},\mathcal{E})$.* □

We are now in a position to give the declarative semantics of higher-order logic program with equality as a natural extension of the declarative semantics of the traditional first-order logic programs.

Theorem 7.16 *There is a least $H/\mathcal{E}$-model $\mathsf{M}_{(\mathcal{P},\mathcal{E})}$ of $(\mathcal{P},\mathcal{E})$, and for all $A \in \Pi(H)$,*

$$(\mathcal{P},\mathcal{E}) \models A \Longleftrightarrow \mathsf{M}_{(\mathcal{P},\mathcal{E})} \models A.$$

□

8 General Programs

In this and following sections, we study various aspects of negation. Since only positive information can be a logical consequence of a definite program, special rules are needed to deduce negative information. The most important of these rules are the closed world assumption and the negation as failure rule. Next section introduces general programs, which are programs for which the body of a program clause can contain negation symbol. The major results of this paper are soundness theorems for the negation as failure rule and SLDNF-resolution for general programs.

The framework of definite clauses presented before allows us to obtain only "positive information", i.e. the only goals which are logical consequences are positive. The lack of ability to obtain "negative information" is a major drawback from both the theoretical and practical point of view. In dealing with models of logic formulas in general, there is duality between both truth values. In practice, this duality can be extremely important, for example in database applications

There are two main approaches to this problem. The first is to extend the language of definite clauses. For example, one familiar extension used in Prolog systems is that of clauses containing at least one positive literal. Known colloquially as "negation in the body", this extends definite clauses, which are clauses containing exactly one positive literal.

The second approach is to adopt special rules or assumptions which tell us, under given circumstances, when information is negative. Amongst the most prominent of these are the closed world assumption and the negation as failure rule. The first states that all atoms which are not logical consequences are false. The second is implementation dependent; it states that an atom is false if all attempts to prove it terminate unsuccessfully.

Our approach is a combination of both these approaches. Based on the concept of completed databases and the negation as failure rule of Clark [10], our complete logic programs, written $(\mathcal{P}^*, \mathcal{E}^*)$, allow us to have negative goals as logical consequences, whereas a definite clause program $(\mathcal{P}, \mathcal{E})$ can not. From an operational point of view, we adopt a negation as failure rule. We justify our approach by showing that these declarative and operational aspects of negation coincide.

9 Programming with the Completion

In this section, general programs are introduced. These are programs whose program clauses may contain negation symbols in their body. The completion of a program is also defined. The completion will play an important part in the soundness results for the negation as failure rule and SLDNF-resolution. The definition of a correct answer is defined for general programs.

A formal definition of complete logic programs requires the concept of unification completeness of an equality theory.

We now define generalized unification over an equality theory $\mathcal{E}$. An *$\mathcal{E}$-unifier* of two terms s and t is a substitution θ such that $\mathcal{E} \models \theta s = \theta t$. An important property is that two terms are $\mathcal{E}$-unifiable iff there is a closed substitution over H of the terms such that the closed instances are both in the same class of the finest congruence over H generated by $\mathcal{E}$.

This does not mean, however, that if two terms are equal in another algebra modelling $\mathcal{E}$ then they are $\mathcal{E}$-unifiable.

Let $\overline{s} = s_1, \cdots, s_n$ and $\overline{t} = t_1, \cdots, t_n$ be two sequences of terms of length n, and write $\overline{s} = \overline{t}$ for

$$x_1 = t_1 \wedge \cdots \wedge x_n = t_n.$$

We already have, by definition, an intimate connection between truth in $\mathcal{E}$ and $\mathcal{E}$-unification; two sequences of terms $\overline{s}$ and $\overline{t}$ are $\mathcal{E}$-unifiable iff $\mathcal{E} \models \exists\overline{x}(\overline{s} = \overline{t})$. With negation issue at hand, we need a dual property; that is, we need to establish a relationship between non-existence of $\mathcal{E}$-unifiers and falsity in $\mathcal{E}$. We thus require that an equality theory dictates that equality holds only if $\mathcal{E}$-unification is possible. To express this formally, if θ is the substitution

$$[t_1/x_1, \cdots, t_n/x_n]$$

let $eqn(\theta)$ denote the conjunction of equations

$$x_1 = t_1 \wedge \cdots \wedge x_n = t_n.$$

For each pair $\overline{s}, \overline{t}$ of sequences we require the existence of a set $U(\overline{s}, \overline{t})$, possibly empty, possibly infinite, of $\mathcal{E}$-unifiers such that if $\overline{y} = y_1, \cdots, y_k$ are all free variables in $\overline{s}$ and $\overline{t}$ then

$$\mathcal{E} \models \overline{s} = \overline{t} - \bigvee_{\theta \in U(\overline{s}.\overline{t})} \exists eqn(\theta)$$

where $\exists$ denote existential quantification of those free variables in $eqn(\theta)$ which are not in $\overline{y}$. We adopt the convention that an empty disjunction is false. Thus the above expression means that if an assignment of the free variables in terms $\overline{s}$ and $\overline{t}$ is such that $\overline{s} = \overline{t}$ is true in a model of $\mathcal{E}$, then at least for one of the $\mathcal{E}$-unifiers θ, $\exists eqn(\theta)$ is also true in the same model and assignment. Consequently, when there is no unifiers of $\overline{s}$ and $\overline{t}$ (i.e. $U(\overline{s}, \overline{t}) = \emptyset$), $\mathcal{E} \models \overline{s} \neq \overline{t}$.

The essence of unification completeness is that every possible solution of any given equation can be represented by an $\mathcal{E}$-unifier of the equation. In particular, when there are no $\mathcal{E}$-unifiers, there can be no solution.

Let $\overline{s}$ and $\overline{t}$ be two sequences of terms of equal length, and σ and θ two $\mathcal{E}$-unifiers of $\overline{s}$ and $\overline{t}$. Then we say that σ is a *more general $\mathcal{E}$-unifier than θ*, denoted by $\sigma \leq \theta$ iff σ is a more general substitution than θ is. An $\mathcal{E}$-unifier σ is *maximal* iff there is no $\mathcal{E}$-unifier which is more general than σ.

Next we extend the definition of goal formula to that of general goal formula to incorporate the negation symbol.

Definition 9.1 A *general goal formula* is defined inductively as follows:

(a) An equation of the form $s_\alpha = t_\alpha$ where $s_\alpha, t_\alpha \in T(\Sigma)$ is a general goal formula.

(b) $\top$ is a general goal formula.

(c) An atomic goal A is a general goal formula.

(d) If G_1 and G_2 are general goal formulas, then so are $G_1 \vee G_2$, $G_1 \wedge G_2$, $\exists x G_1$ and $\neg G_1$.

Note that atomic goal formulas and terms s_α, t_α in equation $s_\alpha = t_\alpha$ do not contain symbols $=$ and $\neg$.

We shall use the following abbreviations:

1. $\forall x G$ for $\neg\exists x \neg G$
2. $G_1 \supset G_2$ for $(\neg G_1) \vee G_2$
3. $s \neq t$ for $\neg(s = t)$.

□

Definition 9.2 A *general program clause* is a clause of the form

$$A \leftarrow G$$

where A is a rigid atom and G is a general goal formula. We call A the *head* of clause and G the *body* of clause. □

In [19] normal programs in first-order logic are defined. These are programs whose program clauses my contain negative "literals" in their body. In higher-order logic, however, normal programs are meaningless; if atom of a negative literal is flexible then by substituting a term for the head predicate variable of the atom we have a general negative goal formula which is not literal. For example, let $\neg Pa$ be a negative literal where P is a predicate variable. By applying the substitution $[\lambda x \cdot px \vee qx/P]$ to this literal we obtain a negative goal formula $\neg(pa \vee qa)$ which is not a literal.

Example 9.3 The well-ordered predicate *wo* can be defined as follows.

$$\begin{array}{l} wo(X) \leftarrow \forall Z(Z \subseteq X \wedge nonempty(Z) \supset hasleastelement(Z)) \\ nonempty(Z) \leftarrow \exists U(Z(U)) \\ hasleastelement(Z) \leftarrow \exists U(Z(U) \wedge \forall V(Z(V) \supset U \leq V)) \\ X \subseteq Y \leftarrow \forall Z(X(Z) \supset Y(Z)) \end{array}$$

□

The increased expressiveness of programs and goals is useful for expert systems, deductive database systems, and general purpose programming applications. In expert systems, it allows the statement of rules in the knowledge base in a form closer to a natural language statement, such as would be provided by a human expert. This makes it easier to understand the knowledge base. In general purpose programming, applications like the above example occur often. If this increased expressiveness is not available it is only possible to express such statement rather obscurely.

Definition 9.4 The *definition* of a predicate constant $p \in \Pi$ in general program $\mathcal{P}$ is the set of all program clauses in $\mathcal{P}$ which have p as top level symbol of their heads. □

Every definite program is a general program, but not conversely.

In order to justify the use of the negation as failure rule, Clark[10] introduced the idea of completion of a general program. We next give the definition of the completion.

Let $p(t_1, \cdots, t_n) \leftarrow G$ be a program clause in a general program $\mathcal{P}$. The first step is to transform the given clause into

$$p(x_1, \cdots, x_n) \leftarrow x_1 = t_1 \wedge \cdots \wedge x_n = t_n \wedge G$$

where $x_1, \cdots, x_n$ are variables not appearing in the clause. Then if $y_1, \cdots, y_m$ are the free variables of the original clause, we transform this into

$$p(x_1, \cdots, x_n) \leftarrow \exists y_1 \cdots \exists y_m (x_1 = t_1 \wedge \cdots \wedge x_n = t_n \wedge G)$$

Now suppose this transformation is made for each clause in the definition of p. Then we obtain $k \geq 1$ transformed formulas of the form

$$p(x_1, \cdots, x_n) \leftarrow E_1$$

$$\vdots$$

$$p(x_1, \cdots, x_n) \leftarrow E_k$$

where each E_i has the general form

$$\exists y_1 \cdots \exists y_m (x_1 = t_1 \wedge \cdots \wedge x_n = t_n \wedge G)$$

and is still a general goal formula. The *completed definition* of p is then the formula

$$p(x_1, \cdots, x_n) \leftarrow E_1 \vee \cdots \vee E_k. \tag{1}$$

Note that $E_1 \vee \cdots \vee E_k$ is also a general goal formula. Some predicate constants in the program may not appear as top level symbol in the head of any program clause. For each such predicate constant q, we explicitly add the clause

$$\neg q(x_1, \cdots, x_n). \tag{2}$$

This is the definition of such q given explicitly by the program. We also call this clause the *completed definition* of such q.

Definition 9.5 An *augmented general logic program $\mathcal{P}^*$ corresponding to program $\mathcal{P}$* is a collection of completed definitions of predicate constants in $\mathcal{P}$. □

In the classical first-order case we form the completion $comp(\mathcal{P})$ of a program $\mathcal{P}$ by taking $\mathcal{P}^*$, the augmented logic program corresponding to program $\mathcal{P}$, and adding the axioms of Clark's equational theory $\mathcal{C}^*$. These axioms, asserting that two terms are equal iff they are unifiable, give a unification complete equality theory, corresponding in a natural way to the standard equality theory $\mathcal{C}$ consisting of the axioms of identity. Thus, for equation e we have $\mathcal{C}^* \models e$ iff $\mathcal{C} \models e$. In general there is no unique way of extending an equality theory $\mathcal{E}$ to a unification complete one $\mathcal{E}^*$

having this relation to it so we can no longer speak of *the* completion of a program $\mathcal{P}$ with equality theory $\mathcal{E}$ but must as in [16] consider a pair $(\mathcal{P}^*, \mathcal{E}^*)$ where $\mathcal{P}^*$ is an augmented general program corresponding to program $\mathcal{P}$ and $\mathcal{E}^*$ some unification complete equality theory. (If $\mathcal{P}$ is thought of as having some underlying equality theory $\mathcal{E}$, then we would require $\mathcal{E}^* \models e$ iff $\mathcal{E} \models e$, but since this does not specify $\mathcal{E}^*$ completely it is presumably $\mathcal{E}^*$ which is directly given.) ¿From now on we always use $(\mathcal{P}^*, \mathcal{E}^*)$ in this sense.

Definition 9.6 For a given set Σ of constants and equality theory $\mathcal{E}$, the ***$\mathcal{E}$-unification problem*** in the language $\mathcal{L}(\Sigma)$ is to decide, for arbitrary terms $s, t \in T(\Sigma)$, whether the set $U(s,t)$ of $\mathcal{E}$-unifiers of s and t is non-empty. The *n^{th}-order $\mathcal{E}$-unification problem* is the $\mathcal{E}$-unification problem for an arbitrary language of order n. If an equational theory $\mathcal{E}$ does not contain other equational clause than the axioms of identity then we write $\mathcal{C}$ for $\mathcal{E}$ and write just unifier or unification for $\mathcal{C}^*$-unifier or $\mathcal{C}^*$-unification respectively. □

For example, the first-order unification problem is known to be decidable. Unfortunately, this does not hold for higher-orders or under general equality theory.

Theorem 9.7 *The second-order unification problem is undecidable.* □

This result was shown by Goldfarb[13] using a reduction from Hilbert's Tenth problem. This result shows that there are second-order (and therefore arbitrarily higher-order) languages where unification is undecidable.

Besides undecidability of $\mathcal{E}$-unification, another problem is that *mgu*'s may no longer exists, a result first shown in [14].

Example 9.8 The two terms $F(a)$ and a have the unifiers $[\lambda xa/F]$ and $[\lambda xx/F]$, but there is no unifier more general than both of these. □

This leads us to extend the notion of a mgu to the $\mathcal{E}$-unification case by considering *complete set of $\mathcal{E}$-unifiers.*

Definition 9.9 Given two sequences of terms, $\overline{s}$ and $\overline{t}$, and a finite set W of variables, a set S of substitutions is a *complete set of $\mathcal{E}$-unifiers of $\overline{s}$ and $\overline{t}$ away from W* (which we shall abbreviate by $CSU(\overline{s},\overline{t})[W]$) iff

1. For all $\sigma \in S$, $Dom(\sigma) \subseteq FV(\overline{s},\overline{t})$ and $Intr(\sigma) \cap (W \cup Dom(\sigma)) = \emptyset$.
2. $S \subseteq U(\overline{s},\overline{t})$.
3. For every $\theta \in U(\overline{s},\overline{t})$, there exists some $\sigma \in S$ such that $\sigma \leq \theta[FV(\overline{s},\overline{t})]$.

When W is not significant, we drop the notation $[W]$. □

Example 9.10 The following set $\mathcal{C}^*$ of equality theory is corresponding to the equality theory $\mathcal{C}$ and $\mathcal{C}^*$ is unification complete. All terms appearing in $\mathcal{C}^*$ are in η-expanded form.

1. $\lambda\overline{x} \cdot F\overline{x} \neq t$ where the term t is rigid and $F \in FV(t)$.
2. $\lambda\overline{x} \cdot f(\overline{s}) \neq \lambda\overline{x} \cdot g(\overline{t})$ where f and g are two different constants.
3. $\lambda\overline{x} \cdot x_i(\overline{s}) \neq \lambda\overline{x} \cdot x_j(\overline{t})$ where $\overline{x}$ is a list of variables of length k and $i, j \in [k]$ such that $i \neq j$.
4. $s_i \neq t_i \rightarrow \lambda\overline{x} \cdot f(\overline{s}) \neq \lambda\overline{x} \cdot f(\overline{t})$ where $\overline{s}$ and $\overline{t}$ are two lists of terms of same length n and $i \in [n]$ and f is a constant.
5. $s_i \neq t_i \rightarrow \lambda\overline{x} \cdot x_j(\overline{s}) \neq \lambda\overline{x} \cdot x_j(\overline{t})$ where $\overline{s}$ and $\overline{t}$ are two lists of terms of same length n and $i \in [n]$ and $\overline{x}$ is alist of variables of length k and $j \in [k]$.

As usual the free variables in equality clause are implicitly universally quantified. Note that a naive extension of Clark's equality theory to higher-order equality theory does not work. For example clause 1 corresponds to clause 4 of Clark's equality theory presented in page 79 of [19]. These clauses are needed because of occur check in unification algorithms. But in higher-order case the two non-convertible terms X and FX are unifiable, since there is a unifier $[\lambda y \cdot y/F]$. Note also that both of these terms are flexible. If one of the two terms is rigid then the occur check will also work for higher-order unification. □

To address the operational semantics of complete logic programs, we return to general logic programs. Corresponding to each $(\mathcal{P}^*, \mathcal{E}^*)$, we obtain a logic program $(\mathcal{P}, \mathcal{E})$ as follows. All that we require of the desired $\mathcal{E}$ is that it shares with $\mathcal{E}^*$ the same finest Σ-congruence. There can be many ways of defining such, e.g., $\mathcal{E} = \{e : e \text{ is a closed equation over } H \text{ and } \mathcal{E}^* \models e\}$.

The general logic program $\mathcal{P}$ we obtain from $\mathcal{P}^*$ is defined as follows. For each predicate definition of type (1) in $\mathcal{P}^*$, obtain k definite clauses where k is the number of disjunctions in the definition body. Then if

$$\exists y_1 \cdots \exists y_m (x_1 = t_1 \wedge \cdots \wedge x_n = t_n \wedge G) \tag{3}$$

is one such disjunct, obtain the corresponding general clause

$$p(t_1, \cdots, t_n) - G \tag{4}$$

Note that we do not construct any general clauses from predicate definitions of type (2) in $\mathcal{P}^*$. Thus we defined $(\mathcal{P}, \mathcal{E})$ corresponding to $(\mathcal{P}^*, \mathcal{E}^*)$.

10 Semantics for general programs

In general programs, we have to interpret the negation symbol to give the definition of satisfaction.

Definition 10.1 Let $\mathcal{M} = \langle D, I \rangle$ be an interpretation of $\mathcal{L}$, φ an assignment into $\mathcal{M}$. When F is a formula in $\mathcal{L}$, we write $\mathcal{M} \models F[\varphi]$ to say that $\mathcal{M}$ satisfies F with respect to φ. For all general goal formulas G, G_1, G_2, for each rigid atom A.

1. When $s_\alpha, t_\alpha \in T(\Sigma)_\alpha$, $\mathcal{M} \models s_\alpha = t_\alpha[\varphi]$ iff $\mathsf{V}_\varphi s_\alpha = \mathsf{V}_\varphi t_\alpha$,
   $\mathcal{M} \models s_\alpha \equiv t_\alpha[\varphi]$ iff $\boldsymbol{e}_I(\mathsf{V}_\varphi s_\alpha) = \boldsymbol{e}_I(\mathsf{V}_\varphi t_\alpha)$.

2. $\mathcal{M} \models \top[\varphi]$

3. $\mathcal{M} \models p(t_1, \cdots, t_n)[\varphi]$ iff $\langle \mathsf{V}_\varphi t_1, \cdots, \mathsf{V}_\varphi t_n\rangle \in Ip$ if p is a constant or $\langle \mathsf{V}_\varphi t_1, \cdots, \mathsf{V}_\varphi t_n\rangle \in \varphi \circ e_I(p)$ if p is a variable

4. $\mathcal{M} \models G_1 \vee G_2[\varphi]$ iff $\mathcal{M} \models G_1[\varphi]$ or $\mathcal{M} \models G_2[\varphi]$

5. $\mathcal{M} \models G_1 \wedge G_2[\varphi]$ iff $\mathcal{M} \models G_1[\varphi]$ and $\mathcal{M} \models G_2[\varphi]$

6. $\mathcal{M} \models \exists x_\alpha G$ iff there is a $d \in D_\alpha$ such that $\mathcal{M} \models G[\varphi[d/x_\alpha]]$

7. $\mathcal{M} \models \neg G_1[\varphi]$ iff $\mathcal{M} \not\models G_1[\varphi]$.

8. $\mathcal{M} \models A \leftarrow G[\varphi]$ iff $\mathcal{M} \models A[\varphi]$ if $\mathcal{M} \models G[\varphi]$.

We write $\mathcal{M} \models F$ to say that a formula F is *valid* in $\mathcal{M}$ if $\mathcal{M} \models F[\varphi]$ for all assignments φ into $\mathcal{M}$. Given a general program $\mathcal{P}$, we say that $\mathcal{M}$ is a *model* or *D-model* for $\mathcal{P}$, and write $\mathcal{M} \models \mathcal{P}$, if each general clause in $\mathcal{P}$ is valid in $\mathcal{M}$. Given a closed goal formula G, we say that G is a *logical consequence* of $\mathcal{P}$, and write $\mathcal{P} \models G$ if G is valid in all models of $\mathcal{P}$. □

Lemma 10.2 *Let $\mathcal{P}$ be a general program. Then $\mathcal{P}$ is a logical consequence of $\mathcal{P}^*$.*
Proof Let $\langle D, J\rangle$ be a general pre-interpretation and I a D-interpretation such that I is a model for $\mathcal{P}^*$. We want to show that I is also a model for $\mathcal{P}$. Let $p(t_1, \cdots, t_n) \leftarrow G$ be a general clause in $\mathcal{P}$ whose free variables are $y_1, \cdots, y_m$, and φ be an assignment into D such that $I \models G[\varphi]$. Assume $\mathsf{V}_\varphi t_i = d_i$ for $i \in [n]$. We need to show that $p(d_1, \cdots, d_n) \in I$.

Consider the completed definition of p

$$p(x_1, \cdots, x_n) \leftarrow E_1 \vee \cdots \vee E_k$$

and suppose E_i is

$$\exists y_1 \cdots \exists y_m(x_1 = t_1 \wedge \cdots \wedge x_n = t_n \wedge G).$$

Let the assignment φ' into D be $(\varphi[d_1/x_1]\cdots[d_n/x_n])$. Then for each $i \in [n]$, $\mathsf{V}_{\varphi'} t_i = d_i$ and $I \models G[\varphi']$, since x_j's do not occur in G. Therefore

$$I \models x_1 = t_1 \wedge \cdots \wedge x_n = t_n \wedge G[\varphi']$$

and $p(d_1, \cdots, d_n) \in I$. □

We can define T operator as in Section 7. Note that T operator for general program is generally not monotonic. For example, if $\mathcal{P}$ is the program

$$p \leftarrow \neg p$$

then $\mathsf{T}_{(\mathcal{P},\mathcal{E})}$ is not monotonic. However, if $\mathcal{P}$ is a definite program, then it is monotonic.

Lemma 10.3 *Let $\mathcal{P}$ be a general program and I be a $D/\mathcal{E}$-interpretation. Then I is a model for $(\mathcal{P}, \mathcal{E})$ iff $\mathsf{T}_{(\mathcal{P},\mathcal{E})}(I) \subseteq I$.*
Proof $\Rightarrow$) Assume $p(d_1, \cdots, d_n) \in \mathsf{T}_{(\mathcal{P},\mathcal{E})}(I)$ for some $p(d_1, \cdots, d_n) \in \Pi(D/\mathcal{E})$. Then there are an assignment φ into $D/\mathcal{E}$ and a clause $p(t_1, \cdots, t_n) \leftarrow G \in \mathcal{P}$ such that $\mathsf{V}_\varphi t_i = d_i$ for all $i \in [n]$ and $I \models G[\varphi]$. Then since $I \models \mathcal{P}$, $I \models p(t_1, \cdots, t_n)[\varphi]$. Therefore $p(d_1, \cdots, d_n) \in I$.
$\Leftarrow$) Similarly. □

Since model intersection property is closely related with monotonicity, model intersection property does not hold as following example shows.

Example 10.4 Let $\mathcal{P}$ be the program

$$p \leftarrow q \wedge \neg r$$
$$q \leftarrow \neg r.$$

Then $\{p, q\}$ and $\{p, r\}$ are models of $\mathcal{P}$. But their intersection $\{p\}$ is not a model of $\mathcal{P}$. □

The next result shows that fixpoints of $\mathsf{T}_{(\mathcal{P},\mathcal{E})}$ give models for $(\mathcal{P}^*, \mathcal{E})$.

Lemma 10.5 *Let I be a $D/\mathcal{E}$-interpretation. Then*
I is a fixpoint of $\mathsf{T}_{(\mathcal{P},\mathcal{E})}$ iff I is a model for $(\mathcal{P}^*, \mathcal{E})$.
Proof Let $p \in \Pi$ and recall that there is only one definition of p in $\mathcal{P}^*$. If it is of the form (1), i.e.,

$$p(x_1, \cdots, x_n) \leftarrow E_1 \vee \cdots \vee E_k,$$

then this definition is satisfied by I iff for all assignment φ into $D/\mathcal{E}$ where $\varphi x_i = d_i, i \in [n]$,

$$p(d_1, \cdots, d_n) \in I \iff \text{for some } E_i, \quad \mathsf{V}'_\varphi t_j = d_j, j \in [n] \text{ and } I \models G[\varphi]$$

Since for each E_i there is a definite clause about p in $\mathcal{P}$ and vice versa, this is the same as

$$p(d_1, \cdots, d_n) \in I \iff p(d_1, \cdots, d_n) \in \mathsf{T}_{(\mathcal{P},\mathcal{E})}(I) \text{ for all } p(d_1, \cdots, d_n) \in \Pi(D/\mathcal{E})$$

If, however, the definition of p is of the form (2),

$$\neg p(\overline{x}) \text{ is satisfied by } I \iff p(\overline{d}) \notin I \text{ for all } \overline{d} \in D/\mathcal{E}.$$

By definition of $\mathsf{T}_{(\mathcal{P},\mathcal{E})}$, we have for each such p that for all $D/\mathcal{E}$-interpretations I and all $\overline{d} \in D/\mathcal{E}$,

$$p(\overline{d}) \notin \mathsf{T}_{(\mathcal{P},\mathcal{E})}(I)$$

Hence $(\mathcal{P}^*, \mathcal{E})$ is satisfied by I iff $\mathsf{T}_{(\mathcal{P},\mathcal{E})}(I) = I$. □

11 $(\mathcal{P}, \mathcal{E})$-derivation

In this and next sections we describe a mechanism that determines whether the existential closure of a goal formula is a logical consequence of a set of program clauses. We would like to describe a procedure that conducts a search for an appropriate derivation sequence that is directed in a sense by the given goal formula. We call this procedure a $(\mathcal{P}, \mathcal{E})$-derivation. $(\mathcal{P}, \mathcal{E})$-derivation may be looked upon as a generalization to higher-order context of the notion of $(\mathcal{P}, \mathcal{E})$-derivations that were introduced in [16, 17], and are prevalent in most discussions of first-order logic programs with equality as the extension of SLD-derivations.

Let the symbols $\mathcal{G}, C$ and θ, perhaps with subscripts, denote sets of general goal formulas, general program clauses, and substitutions, respectively. Let us call a finite set of general goal formulas a *goal set*. We then define the relation of being "$(\mathcal{P}, \mathcal{E})$-derived from" between triples of the form $\langle \mathcal{G}, C, \theta \rangle$ that is basic to the definition of a $(\mathcal{P}, \mathcal{E})$-derivation in the following manner.

Definition 11.1 Let $\mathcal{P}$ be a program. We say a triple $\langle \mathcal{G}_2, C_2, \theta_2 \rangle$ is $(\mathcal{P}, \mathcal{E})$-*derived* from the triple $\langle \mathcal{G}_1, C_1, \theta_1 \rangle$ if one of the following situations holds:

1. (*Goal reduction step*) $\theta_2 = \varepsilon$ and there is a goal formula G in goal set $\mathcal{G}_1$ such that
   - **(a)** G is $\top$ and $\mathcal{G}_2 = \mathcal{G}_1 - \{G\}$, or
   - **(b)** G is $G^1 \wedge G^2$ and $\mathcal{G}_2 = (\mathcal{G}_1 - \{G\}) \cup \{G^1, G^2\}$, or
   - **(c)** G is $G^1 \vee G^2$ and, for $i \in [2]$, $\mathcal{G}_2 = (\mathcal{G}_1 - \{G\}) \cup \{G^i\}$, or
   - **(d)** G is $\exists x_\alpha G^1$ and for new variable $y_\alpha \in \Delta_\alpha$ to goal set $\mathcal{G}_1$ it is the case that $\mathcal{G}_2 = (\mathcal{G}_1 - \{G\}) \cup \{[y_\alpha / x_\alpha] G^1\}$.
2. (*Backchaining step*) Let G be a rigid atom in goal set $\mathcal{G}_1$ such that C_2 is a variant $A \leftarrow G'$ of a clause in $\mathcal{P}$ with no variables in common with those in $\mathcal{G}_1$ and θ_2 is an $\mathcal{E}$-unifier of A and G, and $\mathcal{G}_2 = \theta_2((\mathcal{G}_1 - \{G\}) \cup \{G'\})$.
3. Let G be an equation e in goal set $\mathcal{G}_1$ such that there is an $\mathcal{E}$-unifier θ of equation e. Then θ_2 be an $\mathcal{E}$-unifier equation e which is more general than θ. And $\mathcal{G}_2 = \theta_2(\mathcal{G}_1 - \{e\})$.

In each of the above steps the goal formula G is called the *selected goal* in the goal set $\mathcal{G}_1$. □

Definition 11.2 Let $\mathcal{G}$ be a goal set. Then we say that a (finite or infinite) sequence

$$\langle \mathcal{G}_i, C_i, \theta_i \rangle, i = 0, 1, 2, \cdots$$

is a $(\mathcal{P}, \mathcal{E})$-*derivation sequence for* $\mathcal{G}$ just in case $\mathcal{G}_0 = \mathcal{G}, \theta_0 = \varepsilon$, and for each i, $\langle \mathcal{G}_{i+1}, C_{i+1}, \theta_{i+1} \rangle$ is $(\mathcal{P}, \mathcal{E})$-derived from $\langle \mathcal{G}_i, C_i, \theta_i \rangle$. □

We now introduce the concept of a selection rule, which is used to select goals in a $(\mathcal{P}, \mathcal{E})$-derivation.

Definition 11.3 A *selection rule* is a function from a set of goal sets to a set of goals such that the value of the function for a goal set is a goal, called the *selected goal* in that goal set. □

Definition 11.4 A $(\mathcal{P},\mathcal{E})$-derivation sequence $\langle \mathcal{G}_i, C_i, \theta_i \rangle_{0\le i \le n}$ terminates, i.e. is not contained in a longer sequence, if there is no triple $\langle \mathcal{G}_{n+1}, C_{n+1}, \theta_{n+1}\rangle$ which can be $(\mathcal{P},\mathcal{E})$-derived from. If $\mathcal{G}_n$ is empty, or consists solely of flexible atoms, we say that it is a *successfully terminated* sequence. □

Note that if there are any goal formulas in $\mathcal{G}_n$ then they are of the form

$$Pt_1 \cdots t_n$$

where P is a variable whose type is of the form $\alpha_1, \cdots, \alpha_n \to o$. Let $FP(\mathcal{G}_n)$ be the set of such predicate variable P in $\mathcal{G}_n$. Note that if $\mathcal{G}_n$ is empty then so is $FP(\mathcal{G}_n)$ and for any substitution θ, $\theta \uparrow FP(\mathcal{G}_n)$ is an identity substitution.

Definition 11.5 A $(\mathcal{P},\mathcal{E})$-derivation is *fair* if it is either terminated, or for every goal G in the derivation, (some further instantiated version of) G is selected within a finite steps. □

Definition 11.6 A selection rule R is *fair* if every $(\mathcal{P},\mathcal{E})$-derivation using R is fair. □

For each predicate type π we define the wft E_π

$$E_{\alpha_1,\cdots,\alpha_n \to o} = \lambda x_1 \cdots \lambda x_n \cdot \top$$

where x_i is a variable of type α_i for $i \in [n]$. And we define a generalized substitution

$$\Theta = \{\langle y_\pi, E_\pi\rangle : \pi \text{ is a predicate type and } y_\pi \in \Delta_\pi\}.$$

Definition 11.7 A $(\mathcal{P},\mathcal{E})$ derivation sequence $\langle \mathcal{G}_i, C_i, \theta_i \rangle_{0\le i \le n}$ for $\mathcal{G}$ that is a successfully terminated sequence is called a $(\mathcal{P},\mathcal{E})$-*derivation of* $\mathcal{G}$ and

$$(\theta_1 \circ \cdots \circ \theta_n \circ (\Theta \uparrow FP(\mathcal{G}_n))) \uparrow FV(\mathcal{G})$$

is called its *answer substitution.* If $\mathcal{G} = \{G\}$ then we also say that the sequence is a $(\mathcal{P},\mathcal{E})$-*derivation of* G. □

The following defines the success, finite failure, and general failure sets, denoted by $SS(\mathcal{P},\mathcal{E})$, $FF(\mathcal{P},\mathcal{E})$, and $GF(\mathcal{P},\mathcal{E})$ respectively for a given logic program $(\mathcal{P},\mathcal{E})$.

$SS(\mathcal{P},\mathcal{E}) = \{p(\overline{s}) \in \Pi(H)$: there exists a successful $(\mathcal{P},\mathcal{E})$-derivation sequence of $p(\overline{s})\}$

$FF(\mathcal{P},\mathcal{E}) = \{p(\overline{s}) \in \Pi(H)$: for any fair selection rule, there exists a number n such that all $(\mathcal{P},\mathcal{E})$-derivation sequences of $p(\overline{s})$ are finitely failed with length $\le n\}$

$GF(\mathcal{P},\mathcal{E}) = \{p(\overline{s}) \in \Pi(H)$: for any fair selection rule, all $(\mathcal{P},\mathcal{E})$-derivation sequences of $p(\overline{s})$ are finitely failed $\}$

General failure is, in general, different from finite failure because there can be a closed atom which does not have an infinite derivation sequence and yet there is no number n such that all derivation sequences of this atom are finitely failed with length $\le n$. This possibility arises because $\mathcal{E}$ can be such that there is an infinite set of maximally general $\mathcal{E}$-unifiers for some pair of terms s and t.

Example 11.8 Let $\mathcal{E} = \{f(x, f(y,z)) = f(f(x,y),z)\}$, the theory of an associative function. Noting that the equation $f(y,a) = f(a,y)$ has an infinite number of maximally general $\mathcal{E}$-unifiers

$$[a/y], [f(a,a)/y], [f(f(a,a),a)/y], \cdots,$$

the program $\mathcal{P}$

$$\begin{array}{l} p(a) \leftarrow q(f(a,y), f(y,a)) \\ q(x,x) \leftarrow r(x) \\ r(f(a,x)) \leftarrow r(x) \end{array}$$

is such that $FF(\mathcal{P},\mathcal{E}) \neq GF(\mathcal{P},\mathcal{E})$. This is easily verified by considering the initial goal $p(a)$. So

$$p(a) \in GF(\mathcal{P},\mathcal{E})$$

$$p(a) \notin FF(\mathcal{P},\mathcal{E}).$$

□

However, if $\mathcal{E}$ is such that for all pairs of terms s and t, there is a finite set of maximally general unifiers which subsumes all the $\mathcal{E}$-unifiers of s and t, then $FF(\mathcal{P},\mathcal{E})$ is identical to $GF(\mathcal{P},\mathcal{E})$. In higher-order case even for the equality theory $\mathcal{C}$ there are some pair of terms s and t for which there is no finite set of maximally general unifiers. So in general $FF(\mathcal{P},\mathcal{E}) \neq GF(\mathcal{P},\mathcal{E})$.

Example 11.9 Let the program $(\mathcal{P},\mathcal{C}^*)$ be such that $\mathcal{P}$ consists of the following clauses

$$\begin{array}{l} p(a) \leftarrow q(F(f(a)), f(F(a))) \\ q(x,x) \leftarrow r(x) \\ r(f(x)) \leftarrow r(x) \end{array}$$

The unifiers of the equation $F(f(a)) = f(F(a))$ are

$$[\lambda y \cdot f^k(y)/F], \quad \text{for } k \in \omega.$$

So

$$p(a) \in GF(\mathcal{P},\mathcal{C}^*)$$

$$p(a) \notin FF(\mathcal{P},\mathcal{C}^*)$$

□

12 SLDNF-resolution with Equality

In this section we define an appropriate version of SLDNF-resolution for higher-order general programs and goals with equality theory and prove, in next section, for it the analogue of Clark's fundamental theorem[10], that if a goal succeeds it is a consequence of the completion of the program, and if it fails then its negation is a consequence. It is to be expected that most of the other properties of SLDNF-resolution could be proved in this more general context, but the results are of

(b) G_m is a closed negated goal $\neg G^1$ and there is an SLDNF-refutation of rank $< \nu$ of $\{G^1\}$, or

(c) G_m is an equation e and e has no $\mathcal{E}^*$-unifier.

(d) G_m is an inequation $s \neq t$ where $\mathcal{E}^* \models s = t$.

Note that an SLDNF-refutation (respectively, generally failed SLDNF-tree) of rank ν is also an SLDNF-refutation (respectively, generally failed SLDNF-tree) of rank μ, for all $\mu \geq \nu$.

Definition 12.2 Let $(\mathcal{P}, \mathcal{E}^*)$ be a general program and G a general goal formula. An SLDNF-refutation of $(\mathcal{P}, \mathcal{E}^*) \uplus \{G\}$ is an SLDNF-refutation for $(\mathcal{P}, \mathcal{E}^*) \uplus \{G\}$ of rank ν, for some ν. □

Definition 12.3 Let $(\mathcal{P}, \mathcal{E}^*)$ be a general program and G a general goal formula. A generally failed SLDNF-tree for $(\mathcal{P}, \mathcal{E}^*) \uplus \{G\}$ is a generally failed SLDNF-tree for $(\mathcal{P}, \mathcal{E}^*) \uplus \{G\}$ of rank ν, for some ν. □

If a goal set contains only flexible atoms and negated atoms which are not closed, then no goal is available for selection. We now formalize this notion. By *computation* of $(\mathcal{P}, \mathcal{E}^*) \uplus \{G\}$, we mean an attempt to construct an SLDNF-derivation of $(\mathcal{P}, \mathcal{E}^*) \uplus \{G\}$.

Definition 12.4 Let $(\mathcal{P}, \mathcal{E}^*)$ be a general program and $\mathcal{G}$ is a general goal set. We say a computation of $(\mathcal{P}, \mathcal{E}^*) \uplus \mathcal{G}$ *flounders* if at some point in the computation a goal set is reached which contains only flexible atoms and negated atoms which are not closed. □

In 2.(a) of the definition of SLDNF-refutation, the transformations for negated formulas have been presented to try to overcome the limitations of the negation as failure rule. For example, without 2.(a).iii), the computation of $(\mathcal{P}, \mathcal{E}^*) \uplus \{\neg\neg G\}$ can flounder if G contains any free variables. This problem disappears once the goal is transformed to G. Similar problems are overcome by 2.(a).i), and ii).

Now that we have given the definition of computed answer, we consider the procedure a logic programming system might use to compute answers. The basic idea is to use $(\mathcal{P}, \mathcal{E}^*)$-derivation, augmented by the negation as failure rule. When a non-negative goal is selected, we use essentially $(\mathcal{P}, \mathcal{E}^*)$-derivation to derive a new goal set. However, when closed negative goal is selected, the goal answering process is entered recursively in order to try to establish the negative subgoal. We can regard these negative subgoals as separate *lemmas*, which must be established to compute the result. Having selected a closed negative goal $\neg G$ in some goal set, an attempt is made to construct a generally failed SLDNF-tree with root $\{G\}$ before continuing with the remainder of the computation. If such a generally failed tree is constructed, then subgoal set $\{\neg G\}$ succeeds. Otherwise, if an SLDNF-refutation is found for $\{G\}$, then the subgoal set $\{\neg G\}$ fails. Note that bindings are only made by successful calls of positive rigid atoms. Negative calls never create bindings; they only succeed or fail. Thus negation as failure is purely a test.

Example 12.5 Let $\mathcal{P}$ consist of the following clauses

$$
\begin{aligned}
&x \subseteq y \leftarrow \forall u(x(u) \supset y(u)) \\
&p(a) \leftarrow \top \\
&q(a) \leftarrow \top \\
&q(b) \leftarrow \top
\end{aligned}
$$

An SLDNF-refutation of $(\mathcal{P}, \mathcal{C}^*) \uplus \{p \subseteq q\}$ is

$$p \subseteq q$$
$$\forall u(p(u) \supset q(u))$$

which succeeds since the final goal is an abbreviation of closed goal $\neg\exists u \neg(\neg p(u) \vee q(u))$ and we can build a failed SLDNF-tree for $\exists u \neg(\neg p(u) \vee q(u))$.

$$\exists u \neg(\neg p(u) \vee q(u))$$
$$\neg(\neg p(u) \vee q(u))$$
$$\neg\neg p(u) \wedge \neg q(u)$$
$$\neg\neg p(u), \neg q(u)$$
$$p(u), \neg q(u)$$
$$\top, \neg q(a)$$
$$\neg q(a)$$

A failed SLDNF-tree for $(\mathcal{P}, \mathcal{C}^*) \uplus \{q \subseteq p\}$ is

$$q \subseteq p$$
$$\forall u(q(u) \supset p(u))$$

which is failed, since the final goal is an abbreviation for $\neg\exists u \neg(\neg q(u) \vee p(u))$ and there is an SLDNF-refutation for $(\mathcal{P}, \mathcal{C}^*) \uplus \{\exists u \neg(\neg q(u) \vee p(u))\}$

$$\exists u \neg(\neg q(u) \vee p(u))$$
$$\neg(\neg q(u) \vee p(u))$$
$$\neg\neg q(u) \wedge \neg p(u)$$
$$\neg\neg q(u), \neg p(u)$$
$$q(u), \neg p(u)$$
$$\top, \neg p(b)$$
$$\neg p(b)$$

which succeeds because of the failed tree.

$$p(b)$$

□

13 Soundness of SLDNF-derivation with Equality

Let $\mathcal{G}$ be a general goal set $\{G_1, \cdots, G_n\}$ which occurs in a place where normally a formula can do. Then by $\mathcal{G}$ we mean the conjunction $G_1 \wedge \cdots \wedge G_n$. And we adopt the convention that empty conjunction is true.

Theorem 13.1 *If $\mathcal{P}$ is a general program, $\mathcal{E}^*$ a unification complete equality theory and $\mathcal{G}$ is a general goal set, then for all ordinals ν*

(a) *if* $(\mathcal{P}, \mathcal{E}^*) \uplus \mathcal{G}$ *has a generally failed SLDNF-tree of rank* ν *then* $(\mathcal{P}^*, \mathcal{E}^*) \models \neg\mathcal{G}$.

(b) *if* $(\mathcal{P}, \mathcal{E}^*) \uplus \mathcal{G}$ *has an SLDNF-refutation of rank* ν *with answer* θ *then* $(\mathcal{P}^*, \mathcal{E}^*) \models \theta\mathcal{G}$.

Proof We prove these simultaneously by induction on ν.
(a) We prove the contrapositive, that if there is a $H/\mathcal{E}^*$-model I for $(\mathcal{P}^*, \mathcal{E}^*)$ in which $\exists(G)$ is true, then $(\mathcal{P}, \mathcal{E}^*) \uplus \{G\}$ cannot have a generally failed tree of rank ν.

We do this by showing that if an existential closure of a node goal set $\mathcal{G}$ in such a tree is true in I then so is for some successor node goal set $\mathcal{G}'$, which implies the existence of an infinite branch, contrary to the definitions of generally failed tree. Note that an existential closure of a goal set $\{G_1, \cdots, G_p\}$ is true in I means there is some assignment φ into $H/\mathcal{E}^*$ such that

$$I \models G_1 \wedge \cdots \wedge G_p[\varphi].$$

So there is an assignment φ such that for each goal G_i in the goal set,

$$I \models G_i[\varphi].$$

If the selected goal G_m in the node goal set $\mathcal{G}$ is $G^1 \wedge G^2$ then by hypothesis G^1 and G^2 are true in I. So there is a unique child goal set $\mathcal{G}'$

$$(\mathcal{G} - \{G^1 \wedge G^2\}) \cup \{G^1, G^2\}$$

all goals of which are true in I.

If the selected goal G_m in the node goal set $\mathcal{G}$ is $G^1 \vee G^2$ then by hypothesis G^i is true in I for some $i \in [2]$. So there is a child goal set $\mathcal{G}'$

$$(\mathcal{G} - \{G^1 \vee G^2\}) \cup \{G^i\}$$

all goals of which are true in I.

If the selected goal G_m is $\exists y G^1$, then by hypothesis G^1 is true in I under assignment $\varphi[d/y]$ for some $d \in H/\mathcal{E}^*$. There is a unique child goal set $\mathcal{G}'$

$$(\mathcal{G} - \{\exists y G^1\}) \cup \{[z/y]G^1\}$$

where z is a new variable to $\mathcal{G}$. So all of goals in $\mathcal{G}'$ are true in I under assignment $\varphi[d/z]$.

If the selected goal G_m in the node goal set $\mathcal{G}$ is $\neg(G^1 \wedge G^2)$ then there is a unique child goal set $\mathcal{G}'$

$$(\mathcal{G} - \{\neg(G^1 \wedge G^2)\}) \cup \{(\neg G^1) \vee (\neg G^2)\}$$

all of whose goals are true in I under φ, since $(\neg G^1) \vee (\neg G^2)$ is implied by $\neg(G^1 \wedge G^2)$.

If the selected goal G_m is $\neg(G^1 \vee G^2)$ or $\neg\neg G^1$, then the proofs are similar to above case.

If the selected goal G_m in the node goal set $\mathcal{G}$ is a negated closed goal $\neg G^1$ then by hypothesis G^1 is false in I, so by (b) of induction hypothesis $\{G^1\}$ cannot have refutation of rank $< \nu$, so this cannot be a leaf node. So there is a unique child goal set which simply omits $\neg G^1$ and is also true in I.

If the selected goal is an equation $s = t$ then, since this is true in I under assignment φ, and $\mathcal{E}^*$ is unification complete, there is some $\mathcal{E}^*$-unifier θ of s and t such that $\exists\overline{h}eqn(\theta)$ is true under the assignment φ where $\overline{h}$ are the variables in θ not in s or t. Clearly these variables may be chosen different from $\overline{x}$ so that $eqn(\theta)$ is true in I under assignment $\varphi[\overline{d}/\overline{h}]$ for some $\overline{d} \in H/\mathcal{E}^*$. Now $eqn(\theta)$ implies

$$x = \theta x, \quad \text{for each variable } x,$$

hence

$$F \leftrightarrow \theta F, \quad \text{for each formula } F.$$

So all the goals in

$$\theta(\mathcal{G} - \{s = t\})$$

are true in I under $\varphi[\overline{d}/\overline{h}]$. The given node has a child goal set $\mathcal{G}'$

$$\theta'(\mathcal{G} - \{s = t\})$$

for some θ' more general than θ, i.e. such that there is a substitution σ satisfying

$$\mathcal{E}^* \models \theta = \theta' \circ \sigma.$$

Now if σF is true for some assignment ψ then F is true for some variable assignment (viz. the assignment $\sigma \circ \mathsf{V}'_{\psi}$). So all of goals in $\theta'(\mathcal{G} - \{s = t\})$ are true in I under some variable assignment.

If the selected goal is an inequation $s \neq t$ then since this is true in I under variable assignment φ the node cannot be a leaf node, since that requires $\mathcal{E}^* \models s = t$. So it has a unique child goal set $\mathcal{G}'$

$$\mathcal{G} - \{s \neq t\}$$

all of goals of which are also true in I under variable assignment φ.

The last and main case is where the selected goal G_m is a rigid atom $p(\overline{s})$. The completed definition of the predicate p in $\mathcal{P}^*$ is of the form

$$p(\overline{z}) \leftarrow E_1 \vee \cdots \vee E_k$$

where each E_i is of the form

$$\exists\overline{y}(\overline{z} = \overline{t} \wedge G)$$

corresponding to a program clause

$$p(\overline{t}) \leftarrow G$$

where $\overline{z}$ are new variables not occurring in any such clause, and $\overline{y}$ are the free variables of the clause. It is easily seen that the same completed definition of p is obtained whatever variants of the program clauses are used, so we may assume that the same variants are used as are chosen in verifying the definition of generally failed SLDNF-tree at this node so that the variables $\overline{y}$ are distinct from the variables $\overline{x}$. Since $p(\overline{s})$ is true in I under φ one of the formulas

$$\exists\overline{y}(\overline{s} = \overline{t} \wedge G)$$

must be true, i.e. since the variables $\overline{y}$ are distinct from the variables $\overline{x}$,

$$\overline{s} = \overline{t} \wedge G$$

is true in I under a variable assignment $\varphi[\overline{d}/\overline{y}]$ for some $\overline{d} \in H/\mathcal{E}^*$. Since $\mathcal{E}^*$ is unification complete, $\overline{s} = \overline{t}$ implies the existence of some $\mathcal{E}^*$-unifier θ of s and t. As above this implies

$$\theta((\mathcal{G} - \{p(\overline{s})\}) \cup \{G\})$$

is true in I under a variable assignment $\varphi[\overline{d}/\overline{y}][\overline{e}/\overline{h}]$ where $\overline{h}$ are new variables of θ (chosen distinct from $\overline{x}, \overline{y}$). Now by the definition of generally failed tree, since θ unifies $p(\overline{s})$ and $p(\overline{t})$ there must be an $\mathcal{E}^*$-unifier θ' of $p(\overline{s})$ and $p(\overline{t})$ and a substitution σ such that $\mathcal{E}^* \models \theta = \theta' \circ \sigma$ and a child goal set

$$\theta'((\mathcal{G} - \{p(\overline{s})\}) \cup \{G\}).$$

As in the last case, all these goals are true in I for some variable assignment.

(b) This is proved by induction on the length l of the refutation.

l is zero. Then $\mathcal{G}$ is empty or consists only of flexible atoms. If $\mathcal{G}$ is empty, i.e. true, then $(\mathcal{P}^*, \mathcal{E}^*) \models \mathcal{G}$. If $\mathcal{G}$ consists only of flexible atoms, then $\theta\mathcal{G}$ is of the form $\top \wedge \cdots \wedge \top$. So obviously $(\mathcal{P}^*, \mathcal{E}^*) \models \theta\mathcal{G}$.

For the inductive step suppose $l > 0$ and G_m is the first selected goal in the node goal set $\mathcal{G}_0 = \mathcal{G}$.

If G_m is of the form $G^1 \wedge G^2$, then θ_1 is the identity substitution ε and the unique child goal set $\mathcal{G}_1$ is

$$(\mathcal{G}_0 - \{G^1 \wedge G^2\}) \cup \{G^1, G^2\}.$$

By induction hypothesis on l

$$(\mathcal{P}^*, \mathcal{E}^*) \models (\theta_2 \circ \cdots \circ \theta_l \circ (\Theta \uparrow FP(\mathcal{G}_l)))\mathcal{G}_1.$$

Since $\theta_1\{G^1, G^2\}$ implies $\theta_1\{G^1 \wedge G^2\}$, so

$$(\mathcal{P}^*, \mathcal{E}^*) \models (\theta_1 \circ \cdots \circ \theta_l \circ (\Theta \mid FP(\mathcal{G}_l)))\mathcal{G}_0.$$

If G_m is of the form $G^1 \vee G^2$, then θ_1 is the identity substitution ε and next goal set $\mathcal{G}_1$ is

$$(\mathcal{G}_0 - \{G^1 \vee G^2\}) \cup \{G^i\}$$

for some $i \in [2]$. By induction hypothesis on l

$$(\mathcal{P}^*, \mathcal{E}^*) \models (\theta_2 \circ \cdots \circ \theta_l \circ (\Theta \mid FP(\mathcal{G}_l)))\mathcal{G}_1.$$

Since $\theta_1\{G^i\}$ implies $\theta_1\{G^1 \vee G^2\}$, so

$$(\mathcal{P}^*, \mathcal{E}^*) \models (\theta_1 \circ \cdots \circ \theta_l \circ (\Theta \mid FP(\mathcal{G}_l)))\mathcal{G}_0.$$

If G_m is of the form $\exists y G^1$, then θ_1 is the identity substitution ε and the unique child goal set $\mathcal{G}_1$ is

$$(\mathcal{G}_0 - \{\exists y G^1\}) \cup \{[z/y]G^1\}$$

where z is a new variable to $\mathcal{G}_0$. By induction hypothesis on l

$$(\mathcal{P}^*, \mathcal{E}^*) \models (\theta_2 \circ \cdots \circ \theta_l \circ (\Theta \uparrow FP(\mathcal{G}_l)))\mathcal{G}_1.$$

Since $\theta_1\{[z/y]G^1\}$ implies $\theta_1\{\exists y G^1\}$, so

$$(\mathcal{P}^*, \mathcal{E}^*) \models (\theta_1 \circ \cdots \circ \theta_l \circ (\Theta \uparrow FP(\mathcal{G}_l)))\mathcal{G}_0.$$

If G_m is a rigid atom then there is a variant $A \leftarrow G$ of a program clause and $\mathcal{E}^*$-unifier θ_1 of G_m and A, and the next goal set $\mathcal{G}_1$ is

$$\theta_1((\mathcal{G}_0 - \{G_m\}) \cup \{G\}).$$

By induction hypothesis on l

$$(\mathcal{P}^*, \mathcal{E}^*) \models (\theta_2 \circ \cdots \circ \theta_l \circ (\Theta \uparrow FP(\mathcal{G}_l)))\mathcal{G}_1.$$

But

$$\mathcal{P}^* \models (\theta_1 \circ \cdots \circ \theta_l \circ (\Theta \uparrow FP(\mathcal{G}_l)))(A \leftarrow G)$$

and

$$\mathcal{E}^* \models \theta_1 G_m = \theta_1 A,$$

hence

$$(\mathcal{P}^*, \mathcal{E}^*) \models (\theta_1 \circ \cdots \circ \theta_l \circ (\Theta \uparrow FP(\mathcal{G}_l)))\mathcal{G}_0$$

as required.

If G_m is an equation $s = t$ then the next goal set $\mathcal{G}_1$ is

$$\theta_1(\mathcal{G}_0 - \{s = t\})$$

where θ_1 is an $\mathcal{E}^*$-unifier of s and t. By induction hypothesis on l

$$(\mathcal{P}^*, \mathcal{E}^*) \models (\theta_2 \circ \cdots \circ \theta_l \circ (\Theta \uparrow FP(\mathcal{G}_l)))\mathcal{G}_1.$$

Since $\mathcal{E}^* \models \theta_1 s = \theta_1 t$, it follows that

$$(\mathcal{P}^*, \mathcal{E}^*) \models (\theta_1 \circ \cdots \circ \theta_l \circ (\Theta \uparrow FP(\mathcal{G}_l)))\mathcal{G}_0$$

as required.

If G_m is an inequation $s \neq t$, then the next goal set $\mathcal{G}_1$ is

$$\theta_1(\mathcal{G}_0 - \{s \neq t\})$$

where $\theta_1 s$ and $\theta_1 t$ are not $\mathcal{E}^*$-unifiable, i.e.

$$\mathcal{E}^* \models \theta_1 s \neq \theta_1 t$$

since $\mathcal{E}^*$ is unification complete. So

$$(\mathcal{P}^*, \mathcal{E}^*) \models (\theta_1 \circ \cdots \circ \theta_l \circ (\Theta \uparrow FP(\mathcal{G}_l)))\mathcal{G}_0$$

as required.

If G_m is of the form $\neg(G^1 \wedge G^2)$, then θ_1 is the identity substitution ε and the unique child goal set $\mathcal{G}_1$ is

$$(\mathcal{G}_0 - \{\neg(G^1 \wedge G^2)\}) \cup \{(\neg G^1) \vee (\neg G^2)\}.$$

By induction hypothesis on l

$$(\mathcal{P}^*, \mathcal{E}^*) \models (\theta_2 \circ \cdots \circ \theta_l \circ (\Theta \uparrow FP(\mathcal{G}_l)))\mathcal{G}_1.$$

Since $\theta_1((\neg G^1) \vee (\neg G^2))$ implies $\theta_1(\neg(G^1 \wedge G^2))$, so

$$(\mathcal{P}^*, \mathcal{E}^*) \models (\theta_1 \circ \cdots \circ \theta_l \circ (\Theta \uparrow FP(\mathcal{G}_l)))\mathcal{G}_0.$$

If G_m is $\neg(G^1 \vee G^2)$ or $\neg\neg G^1$ then the proofs are similar to the above case.

If G_m is a closed negated goal $\neg G^1$ then there is a generally failed SLDNF-tree of rank $< \nu$ for $(\mathcal{P}, \mathcal{E}^*) \uplus \{G^1\}$ and the next goal set $\mathcal{G}_1$ is

$$\mathcal{G}_0 - \{\neg G^1\}.$$

So by (a) of induction hypothesis on ν

$$(\mathcal{P}^*, \mathcal{E}^*) \models \neg G^1.$$

By induction hypothesis on l

$$(\mathcal{P}^*, \mathcal{E}^*) \models (\theta_2 \circ \cdots \circ \theta_l \circ (\Theta \uparrow FP(\mathcal{G}_l)))\mathcal{G}_1.$$

Since θ_1 is the identity substitution ε we obtain

$$(\mathcal{P}^*, \mathcal{E}^*) \models (\theta_1 \circ \cdots \circ \theta_l \circ (\Theta \uparrow FP(\mathcal{G}_l)))\mathcal{G}_0$$

as required. □

14 Conclusion

We have built model theoretic semantics for higher-order logic programming languages and established the least model and least fixpoint semantics for such languages. Two major relevant aspects of classical first-order logic have been model theory and proof theory; model theory corresponds to

specification and declarative notions, proof theory corresponds to operational semantics and implementations. A proof theoretic characterization of higher-order logic programming is well developed in [26, 24, 28]; this characterization is based on the principle that the meaning of a logic program, provided by provability in a logical system, should coincide with its operational meaning, provided by interpreting logical connectives as simple and fixed search instructions. The operational semantics is formalized by the identification of a class of cut-free proofs called *uniform proofs.*

Even though Miller[25] worried about "unquestioned" use of model theory, we believe that model theoretic development for higher-order logic programming is essential; the existence of a declarative definition provides an important yardstick against which the correctness of an implementation can be measured, for example, without it, we would not be able to even state the soundness and completeness theorems. This situation is even more amplified when the soundness of negation as failure is needed to be justified; in order to assess the proof theoretic power of completions, in contrast to the case of models of definite programs, it is not sufficient to restrict here attention to Herbrand models. It is necessary to consider arbitrary models.

There is a well-known philosophical problem [12]; a knowledge and belief operator such as *knows* creates an opaque context and disallows substitution of equals by equals in an opaque context. Our logic programming languages also create a similar problem; i.e. since they include the propositional type in its primitive set of types, they allow such opaque contexts. This situation can be paraphrased, in our own terms, as: extensional identity of arguments does not imply extensional identity of applications of such arguments to an opaque operator. To solve this problem the researchers in Artificial Intelligence proposed to view a concepts as an object of discourse in logic [21]. In this paper we also take the similar position: we argue that intentions rather than extensions should be main objects of domain of discourse in higher-order logic programming.

We showed that higher-order logic programming possesses the unique semantic properties of first-order logic programming such as the least model and least fixpoint semantics, finite failure and negation as failure.

The work of this paper has, thus, achieved a large part of its original objective, namely that of developing a model theoretic semantics for higher-order logic programming languages that has been proved to be so successful for first-order logic programming.

15 Acknowledgements

We wish to thank Prof. Sanchis for pointing out a serious error in the previous version of this paper. We also would like to thank Prof. Dale Miller for the encouragements and helpful suggestions.

References

[1] James H. Andrews. Predicates as parameters in logic programming: A set-theoretic basis. In P. Schroeder-Heister, editor, *Extensions of Logic Programming*, pages 31–47, 1989.

[2] Peter B. Andrews. Resolution in type theory. *The Journal of Symbolic Logic*, 36(3):414–432, 1971.

[3] Peter B. Andrews. General models and extensionality. *The Journal of Symbolic Logic*, 37(2):395–397, 1972.

[4] Peter B. Andrews. General models, descriptions, and choice in type theory. *The Journal of Symbolic Logic*, 37(2):385–394, 1972.

[5] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth through Proof.* Academic Press, 1986.

[6] Mino Bai. General model theoretic semantics for higher-order horn logic programming. In Andrei Voronkov, editor, *Logic Programming and Automated Reasoning: Proceedings of International Conference LPAR'92, St. Petersburg, Russia*, pages 320–331. Springer-Verlag, July 1992.

[7] H. P. Barendregt. *The Lambda Calculus.* North-Holland, 1984.

[8] Weidong Chen, Kichael Kifer, and David S. Warren. Hilog: A first-order semantics for higher-order logic programming constructs. In Ewing L. Lusk and Ross A. Overbeek, editors, *Logic Programming Proceedings of North American Conference*, pages 1090–1114, 1989.

[9] Alonzo Church. A formulation of simple theory of types. *The Journal of Symbolic Logic*, 5:56–68, 1940.

[10] K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, 1978.

[11] D. DeGroot and G. Lindstrom, editors. *Logic Programming: Relations, functions and Equations.* Prentice Hall, 1986.

[12] David R. Dowty, Robert E. Wall, and Stanley Peters. *Introduction to Montague Semantics.* D. Reidel Publishing Company, 1981.

[13] W. Goldfarb. The undecidability of the second-order unification problem. *Theoretical Computer Science*, 13(2):225–230, 1981.

[14] W.E. Gould. *A Matching Procedure for Omega-Order Logic.* PhD thesis, Princeton University, 1966.

[15] Leon Henkin. Completeness of the theory of types. *The Journal of Symbolic Logic*, 15:81–91, 1950.

[16] J. Jaffar, J-L. Lassez, and M. J. Maher. A theory of complete logic programs with equality. *J. Logic Programming*, pages 211–223, 1984.

[17] J. Jaffar, J-L. Lassez, and M. J. Maher. Logic programming language scheme. In D. DeGroot and G.Lindstrom, editors, *Logic Programming:Relations, functions and Equations*, pages 441–467. 1986.

[18] H. Kirchner and W. Wechler, editors. *Algebraic and Logic Programming.* Springer-Verlag, 1990.

[19] John W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987.

[20] Donald W. Loveland. *Automated Theorem Proving: A Logical Basis.* North-Holland, 1978.

[21] John McCarthy and P. J. Hayes. Some philosophical problems from the viewpoint of artificial intelligence. *Machine Intelligence*, 4:463–502, 1969.

[22] Dale A. Miller. *Proofs in higher-order logic.* PhD thesis, Carnegie-Mellon University, 1983.

[23] Dale A. Miller. A compact representation of proofs. *Studia Logica*, 46(4):347–370, 1987.

[24] Dale A. Miller. Abstractions in logic programming. In Peirgiorgio Odifreddi, editor, *Logic and Computer Science*, pages 329 – 359. Academic Press, 1990.

[25] Dale A. Miller. Proof theory as an alternative to model theory. *Newsletter of the Association for Logic Programming*, 4(3):2–3, 1991.

[26] Dale A. Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. Technical report, University of Pennsylvania, 1989.

[27] Dale A. Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[28] Gopalan Nadathur. *A higher-order logic a the basis for logic programming.* PhD thesis, University of Pennsylvania, 1986.

[29] J. A. Robinson. Mechanizing higher-order logic. *Machine Intelligence*, 4:150–170, 1969.

[30] William W. Wadge. Higher-order horn logic programming. In U. Saraswat and K. Ueda, editors, *Proceedings of International Logic Programming Symposium*, pages 289–303, 1991.

The Architecture of an Implementation of λProlog: Prolog/Mali

Pascal Brisset and Olivier Ridoux
IRISA/INRIA
Campus Universitaire de Beaulieu
35042 RENNES Cedex
FRANCE
{brisset,ridoux}@irisa.fr

1 Abstract

λProlog is a logic programming language accepting a more general clause form than standard Prolog (namely hereditary Harrop formulas instead of Horn formulas) and using simply typed λ-terms as a term domain instead of first order terms. Despite these extensions, it is still amenable to goal-directed proofs and can still be given procedural semantics. However, the execution of λProlog programs requires several departures from the standard resolution scheme. First, the augmented clause form causes the program (a set of clauses) and the signature (a set of constants) to be changeable, but in a very disciplined way. Second, the new term domain has a semi-decidable and infinitary unification theory, and it introduces the need for a β-reduction operation at run-time.

MALI is an abstract memory that is suitable for storing the search-state of depth-first search processes. Its main feature is its efficient memory management.

We have used an original λProlog-to-C translation: predicates are transformed into functions operating on several continuations. The compilation scheme is sometimes an adaptation of the standard Prolog scheme, but at other times it has to handle new features such as types, β-reduction and delayed unification.

Two keywords of this implementation are *sharing* and *folding* of representations. Sharing amounts to recognising that some representation already exists and reusing it. Folding amounts to recognising that two different representations represent the same thing and replacing one by the other.

We assume a basic knowledge of Prolog and λProlog.

2 Introduction

The logic programming language λProlog [28, 27, 29, 14, 12, 26, 13, 30] improves greatly on standard Prolog because it features very powerful operations on terms and programs while still giving them a logical semantics. A keyword common to all these features is *scoping.* λ-terms introduce scoping at the term level, explicit quantifications (universal and existential) introduce scoping at the formula level, and the deduction rules for explicit quantification and implication introduce scoping in proofs,

i.e. at a dynamic level. Deduction rules for λProlog are usually given in the framework of sequent proofs.

λProlog requires some implementation effort for being able to compete with Prolog in efficiency (and then in popularity). Another condition for popularity is to overcome the idea that it is a "difficult" language, but this is another story. The initial implementation of λProlog by Miller and Nadathur, and the second one, eLP, by the Ergo Project at Carnegie-Mellon University, were far from being able to compete with Prolog. Since then, a few teams have worked on the implementation of λProlog. As far as we know[1], current teams are Nadathur, Kwon and Wilson at Duke University [20, 19, 34], Jayaraman at the University of Buffalo (formerly with Nadathur), Elliott and Pfenning at CMU [11], Felty and Gunter at Bell Labs, and the authors at Inria.

Other works are done in a similar framework for integrating linear logic and logic programming (Pareschi and Andreoli [4], Hodas and Miller [17]), or higher-order type systems and logic programming (Elliot [10], Pfenning [36, 37]).

We present in this paper the broad lines of our implementation of λProlog: Prolog/Mali. We have implemented λProlog for its own merits, and as a demonstration that memory management issues are a good guide for implementing logic programming systems. Speed was always our second concern.

We assume a knowledge of Prolog and λProlog, their semantics, and their basic algorithms: logical variable, search-stack, unification, λ-unification [18], deduction rules, and uniform proofs [32, 30]. We adopt an architectural presentation: in section 3, we present the kernel subsystem that is in charge of the elementary representation problems, in section 4, we present a software layer which is both a specialisation and an extension of the kernel, finally, in section 5, we present the compilation scheme. We conclude in section 6.

3 MALI

MALI [6, 38] (Mémoire Adaptée aux Langages Indéterministes — memory for non-deterministic languages) can be specified as the abstract data type *stack of mutable first-order terms.* This abstract data type encompasses the representation of the state of every logic programming language that performs a depth-first search in a search-tree.

MALI is the name of a general principle that has several implementations. The name of the implementation we used in Prolog/Mali is *MALIv06.*

We present what MALI brings to the overall system, and, to avoid any ambiguity, what it leaves undone.

3.1 What MALI brings to Prolog/MALI

3.1.1 A data-structure

MALI brings an abstract data-type which we call *MALI's term.* MALI's terms may be described more concretely as graphs with nodes that can be reversibly substituted. MALI's terms are organ-

[1] We thank the committee member who updated our knowledge-base.

ised in a *term-stack* which is itself a term. A collection of node constructors is offered, among them *atoms* (i.e. leaves), *compound nodes* (i.e. cons or tuples), and *levels* (i.e. term-stack constructors). Some of the compound node constructors are called *mutable* constructors, and the terms constructed with them are called *muterms*. Mutable nodes can be subject to *reversible substitutions*, according to a discipline that is close to the substitution of logical variables in Prolog. According to the discipline, muterms, substitutions, and the term-stack are in the same relationship as logical variables, substitutions, and the search-stack of Prolog. For every kind of node constructor, commands and operations exist for creating and reading them, and for accessing their subnodes (if any). Commands also exist for substituting terms for muterms, and for manipulating the term-stack (pushing, popping, and pruning the term-stack).

Every node constructor can be given an elementary typing via the use of *sorts*. This makes it possible to "decompile" the representation of an application term. For instance, Prolog's integers and constants can be both represented by MALI's atoms, which must be discriminated by their sorts.

In the sequel, we note[2] `(le S R N)` a term-stack[3] of sort `S`, top value[4] `R` and substack `N`, `(at S V)` an atom of sort `S` and value `V`, `([m]c0 S)` a [mutable] nullary compound term of sort `S`, `([m]c2 S T1 T2)` a [mutable] binary compound term of sort `S` and subterms `T1` and `T2`, and `([m]tu S N T1 ... Tn)` a [mutable] compound term of sort `S` and `N` subterms `T1` to `Tn`. We use a labelled notation, `label@term` and `label`, to note different occurrences of the same term. A term may have several labels through substitution, `label1@label2@term`. Terms with labels in common share the same representation; they must be compatible up to a substitution. Terms with different labels (or no label) are different even if they have the same notation; to apply a substitution to one has effect on the others only through occurrences of shared subterms.

It should be clear from this short description that *one of* the intended usages of muterms and the term-stack is the implementation of logical variables and of a search-stack. However, this is the only commitment with logic programming, and other usages are possible. MALI knows nothing about the basic mechanisms of Prolog (resolution, unification), or about λProlog's deduction rules and λ-terms.

3.1.2 A memory management

MALI's terms need memory for their representation. This memory is automatically managed in a way that is optimal with respect to the level of knowledge that is available to MALI. The restriction means that application-dependent accessibility properties are not taken into account by MALI. They can be taken into account indirectly by a proper mapping of the application structures onto MALI's terms.

We call *usefulness logic* the relation that describes which run-time data-structures are useful in a given programming language independently from any particular application. The usefulness logic of the core of logic programming is that

[2]The notation is only a convenience for commenting on MALI's term; it is not part of the programming interface.

[3]A *level*, in MALIv06's jargon.

[4]A *root*, in MALIv06's jargon.

Every useful term is accessible from some search-node under the binding environment of the same *search-node.*

To compare, the usefulness logic of the core of functional programming says that

Every useful term is accessible from some root;

binding environments are not mentioned. So, if one uses a functional programming system for implementing a logic programming system and nothing special is done for memory management, the usefulness logic that is actually implemented cannot be more precise than

Every useful term is accessible from some search-node under some binding environment.

It is usually worse and considers the union of all the binding environments.

The two important features of MALI's memory management are *early reset* and *muterm shunting.* Early reset causes substitutions to be undone[5] by the memory manager seeing that some muterm is never accessible when substituted. Muterm shunting means that substitutions, which are created reversible, may be made definitive[6] seeing that some substituted muterm is never accessible when not substituted. These two features are described at great length in the MALIv06 tutorial [38].

Commands exist for controlling memory management: supplying MALI with new memory resources, taking useless resources from MALI, or starting a garbage collection.

3.1.3 Debugging tools

MALI offers debugging tools for assisting a user in the development of an application. Debugging tools allow to check preconditions of commands, to display components of MALI's state, and to trace commands.

It is important that at every level of an architecture (software or hardware) debugging tools are available. It makes the complexity of composing layers tractable. We will not dwell too long on this subject in other sections; it is enough to know that the specialised intermediate machine (see section 4) also has debugging tools for checking a fair use of everything it defines. The Prolog/Mali system also has debugging tools, but the ultimate level is the level of the λProlog applications which should also come with their debugging tools. This is up to the discipline of λProlog users.

3.2 What MALI leaves undone

3.2.1 A memory policy

We distinguish the management of memory inside an application, which aims at improving the use of some memory supplies, and the management of memory at the interface with a host system, which aims at configuring the supplies. We call *memory policy* the set of decisions related to memory supplies. The decisions range from the amount of memory supplied to MALI, the way this

[5]Without waiting for backtracking to undo these substitutions. Hence the name "Early reset".

[6]Roughly, the effect is to collapse chains of substitutions. Hence the name "Muterm shunting".

memory amount evolves, to the amount of computing power dedicated to memory management ($\approx$ the frequence of the calls to the garbage collector).

A memory policy can be very sophisticated because it deals with many interrelated parameters. For instance, it is likely that, in order to diminish the computing power dedicated to memory management, the total memory allocated to MALI must be increased. However, supplying more memory to MALI may decrease the availability of the host system.

Elementary commands for designing a sophisticated memory policy are available in MALI, but no policy is specified.

3.2.2 Application level terms and execution scheme (unification, resolution, ...)

The only commitment of MALI with logic programming is the term-stack and the muterm substitution. Everything remains to be done as for the representation of the data-structures of an application. The implementor must find a mapping from its application terms onto MALI's terms. In λProlog for instance, the representation of simply typed λ-terms, their unification and normalisation must be mapped on MALI's terms, and on procedures using MALI's commands and operations.

The only hint for mapping application terms and their operations is that it is clearly intended that muterms and the term-stack can be used for representing logical variables and a search-stack.

MALI offers an efficient memory management but brings no solution to the time efficiency. The packaging of MALIv06 is designed to hinder as little as possible any effort to yield speed efficiency.

3.2.3 Program representation

MALI has no notion of program. It is not even intended that an application level program should be represented in MALI. This is a totally independent issue.

4 A specialised intermediate machine

We have designed a specialised intermediate machine (SIM[7]), of the level of the WAM [39, 3], for filling parts of the gap between MALI and λProlog.

The SIM is a specialisation of MALI because it forces some interpretation on MALI's terms. It is also an extension of MALI because it defines new notions that have no equivalent in MALI (e.g. unification, continuations). As a specialisation of MALI, the SIM defines specialised node constructors, and commands and operations for creating, reading, and traversing them. As an extension, it defines commands for implementing the new notions for every specialised node constructors they apply to.

The SIM still says nothing of what will be a program, and what decisions have to be made for ensuring an efficient usage of the machine. This is up to the compilation scheme.

We review what the SIM brings to the overall system.

[7]SIM is not a brand name for *this* specialised intermediate machine; it only designates *this* layer in a software architecture using MALI.

4.1 λProlog terms

To choose a representation for terms in the context of λProlog is a new problem because the requirements of logic programming (Prolog technology), of simply typed λ-calculus, and of uniform proofs of hereditary Harrop formulas must be met at the same time.

Prolog technology requires the representation of logical variables and substitutions. It also requires that substitutions be reversible because the search for a proof is done by a depth-first traversal of a search-tree.

Simply typed λ-calculus requires the representation of abstraction and application, the representation of types, and the capability to compute at least long head-normal forms because the unification procedure needs them. To meet the first requirements, long head-normalisation should be reversible too.

Proving hereditary Harrop sequents is required to represent universally quantified variables and to check the correction of signatures. It also requires the handling of implied clauses but this has little to do with our representation of terms.

We only describe our implementation decisions. The reasons for the decisions are discussed in a technical report by the same authors [8], and in the thesis of the first author [7].

4.1.1 Types

One of the differences between Prolog and λProlog is that the terms of λProlog must be typed for λ-unification to be well defined. Huet's procedure deals with simply typed λ-terms, but λProlog extends simple types with type variables (type schemes). This results in *generic polymorphism.*

Follows a sample declaration for polymorphic homogeneous lists and a polymorphic ternary relation on them.

```
kind list     type -> type.
type []       (list A).
type '.'      A -> (list A) -> (list A).
type append   (list A) -> (list A) -> (list A) -> o.
```

The list `[1,2]` can be represented in MALIv06 like

```
(c2 S_LIST (at S_INT 1) (c2 S_LIST (at S_INT 2) (c0 S_NIL))) .
```

The type `A -> (list A) -> (list A)` can be represented in MALIv06 like

```
(c2 S_ARROW
    A@(mc0 S_UNK_T)
    (c2 S_ARROW listA@(tu S_APPL_T 2 (at S_SYMB_T list) A) listA)) .
```

Type unknown `A` is represented as a mutable nullary compound because it must be reversibly substitutable, and it has no other information associated to it. Note the sharing of `(list A)` indicated by the use of label `listA`.

The idea of generic polymorphism in (λ)Prolog is that

> *Types of different occurrences of a constant are independent instances of its type scheme.*
>
> *Types of different occurrences of (any kind of) a variable are equal.*

In (λ)Prolog, one must also choose whether a clause of the program can be selected on grounds of the type of its predicate symbol or not. We have chosen to forbid selecting a clause on these grounds. It means we follow the *definitional genericity principle* [23]:

> *Types of different body occurrences of a predicate constant are independent instances of its type scheme, whereas types of different head occurrences are renamings of the type scheme.*

With this principle, type inference leads to a non-uniform semi-unification problem which has been shown to be undecidable by Kfoury, Tiuryn and Urzyczyn [21]. In our implementation, types of constants (predicative or not) are only checked, and types of (any kind of) variables are inferred.

The reason for sticking to definitional genericity is that it is the most natural when predicates are seen as definitions and type schemes as abstractions of the definitions. It is also required for allowing a simple but sound modular analysis of programs. We want to be able to type-check a module using the type schemes of the modules it imports but not the modules themselves.

In λProlog, it is necessary to represent types at run-time for controlling unification, and some conditions are missing for having a *semantic soundness* result of the kind "Well typed programs cannot go wrong"[8].

The problem wi semantics soundness is that nothing restricts λProlog constants to have the *type preserving property* [15]:

> *Every type variable in a type scheme should appear in the result type (the type to the right of the right-most* `->`*).*

The advantage of having the type preserving property is that the types of the subterms of a term built with a type preserving constant can be inferred from the type of the term. The disadvantage is that it is not flexible enough for representing dynamic types [1].

Types for "not going wrong" We call *forgotten type variables* the type variables that do not occur in the result type of a non-preserving type scheme. We call *forgotten types* the instances of the forgotten type variables. Only forgotten types need to be represented at run-time for avoiding "going wrong". They must be attached as supplementary arguments to the term constructors that are not type preserving. These pseudo-arguments must always be unified before the regular arguments. This makes λ-unification problems always well-typed.

In fact, what is implemented is the representation and unification of terms of a polymorphic type system [5]. It is as if a symbol defined as

```
kind  dummy     type.
type  forget    _ -> dummy.
```

were defined as

```
type  forget_   'PI' A\ (A -> dummy).
```

[8] In this context, "going wrong" means "trying to solve ill-typed unification problems".

where `'PI'` is the product type quantifier, and a term like `(forget 1)` were `(forget_ int 1)`. The term `(forget_ int 1)` can be represented in MALIv06 like

```
(tu S_APPL 3 (at S_SYMB forget_) (at S_SYMB_T int) (at S_INT 1)) .
```

Note that, unlike Typed Prolog [31, 23], there is no special syntax in λProlog for declaring predicate constants. They are only distinguishable by their result type, `o`. So, every predicate constant forgets every type variable in its type because its result type contains no type variable. It can be shown that if the predicates obey the definitional genericity principle, unification of these forgotten types will always succeed; type unification of types forgotten by predicate constants is only required for conveying types along the computation. In a system that does not need that conveying (say, standard Prolog), the forgotten types of predicate constants need not be represented [31, 15]. In λProlog, conveying the types is required for controlling unification.

Types for controlling projection in unification Let us first recall the core of Huet's λ-unification procedure [18].

For a pair $< \lambda\overline{x} \cdot (F\ \overline{s_p}), \lambda\overline{x} \cdot (@\ \overline{t_q}) >$, where F is a logical variable (a *flexible* head) and @ is not a logical variable (a *rigid* head), at most *p+1* substitutions are produced by two rules.

1. If @ is a constant, the *imitation* rule produces $F \leftarrow \lambda\overline{u} \cdot (@\ \overline{E_q})$.

2. For each $0 \leq i \leq p$ such that $\tau(s_i) = \tau_1 \rightarrow \ldots \tau_m \rightarrow \tau((F\ \overline{s_p}))$, the *projection* rule produces $F \leftarrow \lambda\overline{u} \cdot (u_i\ \overline{E_m})$.

Every E_k in $\overline{E}$ stands for $(H_k\ \overline{u})$, where H_k is a new logical variable with the appropriate type.

The projection rule is controlled by a type condition (2. above). For the condition being testable at run-time it is enough that logical variables are equipped with their types.

Note that the types of logical variables themselves need never be unified because when a unification problem is to be solved then it is well-typed (i.e. the two terms of the problem have identical types). This is a side-effect of unifying first the forgotten types in the pseudo-arguments and then the regular arguments.

In λProlog, nothing prevents having a type with a variable result type. This makes the checking of the type condition unsafe: there can be no argument satisfying the condition in some binding environment while projection is possible in a more precise binding environment. The only safe solution is to suspend unification until the result type get known. However, the traditional solution is to commit the result type to be a constant [35]. We believe that nothing satisfactory will be done before these *flexible types* are better understood.

Types for new logical variables It is easy to attach a type to logical variables coming from the program: it is an outcome of the type inference/checking. But the imitation and projection rules of λ-unification introduce new logical variables that correspond to nothing in the source program (the H_k's). They must be attached a type anyway. They all have types $\nu_1 \rightarrow \ldots \rightarrow \nu_p \rightarrow$?? where the ν_i's are the types of the arguments of the flexible head, and ?? depends on the rule.

In case of projection, the ?? of every new logical variable H_j is τ_j (see above in the condition controlling projection). In case of imitation, the ?? is the type of the corresponding argument of the rigid head. So, it remains to be able to infer the types of rigid heads in unification probkems.

There are three kinds of rigid heads: λ-variables (but they cannot be imitated), function constants, and universal variables (they are introduced for solving universally quantified goals). First, we attach their type to every universal variable. Second, we observe that the type scheme of a constant, plus the forgotten types attached to it, plus the result type[9], give enough information for reconstructing the full type of the constant. A type reconstructing function is generated at compile-time from every type scheme declaration. It gets the forgotten types from the constant head and the result type from the flexible head, and it returns the type of the constant head.

4.1.2 λProlog terms

Terms are represented using the full copy technique (as opposed to structure-sharing or a mix of structure-sharing and copy) for memory management reasons: this gives the most precise allocation/deallocation operations for any type of control, and λProlog needs to depart from the standard control.

A novelty of λProlog is that terms need normalisation. In Prolog/Mali, normalisation alters the representation of term for sharing reduction effort, and also for memory management. λ-terms are represented by graphs, and normalisation is implemented as graph-reduction.

Abstractions and applications We will see that logical variables are not the only application level structures that can be represented by MALI's muterms.

Abstractions and applications are represented by reversibly mutable graphs, so that it is possible to physically replace a redex by its reduced form in the graph. This provides sharing of the reduction effort. Reversibly means that mutations (reductions) can be undone when backtracking. This is the result of inserting graph reduction in a Prolog context.

Substituting new representations for older ones in a reversible way forces to store all the history of every term representation. However, MALI's memory management, especially muterm shunting, will remove every useless old representation. Muterm shunting shortens the history of term representations.

Terms are represented as much as possible in their long head-normal form. So, abstractions and applications are in fact tuples of nested elementary abstractions and applications. The term $\lambda nsz \cdot (s\ (n\ s\ z))$ can be represented in MALIv06 like

```
(mtu S_ABST 4
     n@(c0 S_VAR) s@(c0 S_VAR) z@(c0 S_VAR)
     (mtu S_REDEX 2 s (mtu S_REDEX 3 n s z))) .
```

The applications are potential redexes, hence the sort `S_REDEX`.

[9]In the context of unification, it can be found in the flexible head.

First-order terms We call informally *first-order terms* the rigid terms whose head is a constant. They are distinguished as much as possible because they are definitely in long head-normal form and they can be unified by a cheaper procedure.

Universal variables and logical variables In the following, we say that a logical variable *captures* a term if it is bound to a value that contains the term. So, a logical variable is able to capture terms of any type provided they are properly wrapped in a binding value.

Universal variables are among the new constructs of λProlog that enforce checking scoping conditions. A universal variable can be captured by every logical variable of its scope, whereas it cannot be captured outside its scope. I.e. in context $\ldots\forall x\ldots\exists U\ldots\forall y\ldots$, universal variable x can be captured by logical variable U, but y cannot. λ-variables are essentially universal, they are always bound in the rightmost part of the context. So, they can never be captured by logical variables. Constants are also essentially universal, but they are always bound in the leftmost part of the context. So, they can always be captured by logical variables.

Scopes of universal variables are represented by their nesting level. A nesting level is attached to every logical variable and every universal variable, and a register contains the value of the current nesting level. When a universally quantified goal is executed, the nesting level register is first incremented, and then a new universal variable is created with the new nesting level value. Every further creation of logical variables within the scope of this goal but out of the scope of any nested universal quantification will be done with the new nesting level value. We assume that the initial nesting level is 0.

Given that logical variables and universal variables must also carry their types, they can be represented in MALIv06 like

```
(mc2 S_UNK type (at S_SIG nesting_level))
(c2 S_UVAR type (at S_SIG nesting_level))
```

When an attempt is made to substitute a term for a logical variable, the scopes of the term and all its subterms are checked using the nesting levels. If the term contains universal variables of a higher nesting level than the logical variable then the substitution is illegal. If the term contains logical variables of a higher nesting level than the substituted logical variable then their nesting levels should be lowered to the nesting level of the substituted logical variable. If a universal variable or a logical variable with a higher nesting level is in fact in an argument of a flexible term then the scope-checking must be suspended because the problematical universal variable or logical variable may disappear as a side-effect of another substitution. For instance, $X^1 \leftarrow (U^1\ 1\ Y^2)$ is a problematical substitution[10], but after substitution $U^1 \leftarrow \lambda xy \cdot (F^1\ x)$ is applied, it is no more problematical.

We have seen that logical variables cannot capture λ-variable, and can only capture universal variables whose scope they belong to. A flexible term can be seen as a generalisation of the logical variable which is explicitly allowed to capture supplementary terms (the arguments). For instance, the flexible term above is a generalised logical variable that is implicitly allowed to capture the universal variable of level 1 and every constant, and is explicitly allowed to capture 1, which is only

[10]The nesting levels are written as superscripts.

redundant because it wa already implicit, and Y^2, which is not redundant because of the nesting level of U^1.

One of the effects of substituting a term to a logical variable is to diminish the allowance of a generalised logical variable (see the same flexible term after substitution $U^1 \leftarrow \lambda xy \cdot (F^1\ x)$ is applied). Allowance cannot increase because the binding value of a logical variable must be in its allowance.

The main consequence is that no decision related to the occurrence of some patterns can be complete when involving flexible terms. We have exposed it for scope-checking, but it is also true for the occurrence-check in unification ($X \leftarrow t$ is a legal substitution only if $X \notin FV(t)$). If some term has an occurrence in a flexible term, a substitution may take it away.

4.1.3 Reduction

Reduction is implemented as graph-reduction. Since abstractions and applications are not represented one at a time but as tuples, reduction considers simultaneously several β-redexes. This saves term traversing and duplication, hence time and memory.

The basic scheme is to duplicate the left-most part of a redex, and to replace λ-variables occurrences by the arguments. A critical improvement over the basic scheme is to recognise combinators which are subterms of the left-most part of redexes; they need not be duplicated. Every logical variable, every goal argument, and every instance of a term that is a combinator is a combinator. This shows that many terms are combinators and that once a combinator is detected it is safe to tag it as such. Tagging amounts to having more sorts for representing the terms of the cross-product (combinator/non-combinator)×(abstraction/application).

This improvement is fundamental and changes the complexity of useful λProlog predicates [9]. It is not committed to our architecture; it only has to do with reduction.

4.1.4 λ-unification

The now conventional names for the different procedures of λ-unification are SIMPL, MATCH and TRIV. We add UNIF1 and a specialised unification command of the SIM for every kind of term constructors. The main idea is to consider the different unification procedures as as much sieves. If a unification problem cannot be handled by a procedure it is passed to the next one.

Specialised unification commands A sequence of specialised unification commands is generated by the compiler for every clause head. Specialised unification commands can be seen as resulting from a partial evaluation of the general unification procedure. In case there is not enough information in the head (e.g. a second occurrence of a logical variable), the control is passed to procedure UNIF1. This is much like what is done in standard Prolog systems. In case the head term is higher-order, they only build a representation of the unification problem and pass it to SIMPL.

UNIF1 A first-order unification procedure, UNIF1, is used as much as possible on the so-called "first-order terms" (terms with sort `S_APPL`) until a higher-order term is met.

SIMPL When a higher-order term pops up in UNIF1 or in the specialised unification commands, one switchs to procedure SIMPL. The outcome of SIMPL is a set of flexible-rigid pairs which, if it is not empty, is passed to procedure MATCH. If it is empty, a success is reported.

Procedure SIMPL may report a failure if a clash of constants or λ-variables occurs.

MATCH Procedure MATCH is the non-deterministic part of λ-unification. It is described as the core of Huet's procedure in section 4.1.1.

Its non-determinism and the one coming from the proof-search are merged in a single search process. To do the merging easily, we write the control of MATCH in λProlog. Only the great lines of MATCH are written in λProlog: the non-deterministic choice between imitation and projections. The actual imitation and projection rules are implemented as deterministic built-in predicates.

Suspensions Flexible-flexible pairs cannot usually be solved as such because they have too many arbitrary solutions. They are suspended. We use the versatility of MALI's muterms for encoding the suspended flexible-flexible pairs within the flexible heads as a constraint. As soon as one of the flexible heads becomes bound, its constraints are checked. This is similar to the *attributed variable* technique described by Le Huitouze [24].

TRIV A flexible-rigid pair is not passed directly to procedure MATCH, nor is a flexible-flexible pair automatically suspended. They are first passed to procedure TRIV, which tries to solve them in a fast deterministic way. TRIV applies various heuristics; if none works the pair is actually passed to MATCH or suspended.

The heuristics aim at finding pairs of the form $< X, t >$ under various disguises. If such a pair is discovered and logical variable X does not occur in term t then $X \leftarrow t$ is the solution to the unification problem. In a way similar to the scope-checking in section 4.1.2, the occurrence-check is more complicated than for the first-order case because not all occurrences of X in t are dangerous. If one is found and it is dangerous then unification fails. If it is not dangerous then TRIV passes the pair to MATCH.

Some disguises under which a good TRIV procedure must recognise a trivial pair are

1. $< \lambda x.(X\ x), t >$, which is η-equivalent to a trivial pair,
2. and $< X^i\ u^{i+1} \ldots u^{i+j}, t >$, where the superscripts represent the scope nesting, and the u^k's are universal variables; it is equivalent to $< X'^{i+j}, t >$ for a new logical variable X'. In this case, the solution substitution is $X^i \leftarrow \lambda x_1 \ldots x_j \cdot [u^{i+1} \leftarrow x_1] \ldots [u^{i+j} \leftarrow x_j]t$ for taking into account the disguise.

The second disguise is very frequent because a lot of λProlog programming is about exchanging universal variables and λ-variables (i.e. essentially universal quantification at the formula level and essentially universal quantification at the term level). The following predicate is an example of the exchanging trick:

```
type  list2flist  (list A) -> ((list A) -> (list A)) -> o.
list2flist  L  FL :- pi list\(conc  L  list  (FL list)) .
```

The predicate relates the standard representation of a list (say `[1,2,3]`), and its functional representation (`z\ [1,2,3|z]`) [9].

Folding representations The logic of unification is to find a substitution making two terms equal. If they are equal then they can share the same representation. We have seen that both abstractions and non-first-order applications are represented by muterms. So, it is easy to make the two terms share the same representation by substituting one for the other. The effect is to fold the representations because two terms with initially different representations end up to have the same. This substitution must be reversible (like the others: solution substitution and λ-reduction substitution). Reversibility comes as a consequence of using muterms. Folding saves unification effort because identity of representation is much easier to check than equality. It also saves memory, hence garbage collection time.

Terms in unification problems must be in long head-normal form before being compared. After applying the substitutions invented by imitation or projection, the flexible term may be no more in long head-normal form. However, its new long head-normal form is easy to deduce from the term and the substitution without using the β-reducer. So, imitation and projections invent a substitution value, substitute it for the head of the flexible term, compute its new long head-normal form, and substitute it for the flexible term.

For instance, unification problem $< t_1, t_2 >$, where $t_1 = \lambda x \cdot t_3$, $t_3 = (U\ (x\ S_1))$, and $t_2 = \lambda x \cdot (x\ S_2)$, yields three substitutions after one run of MATCH:

1. $U \leftarrow \lambda y \cdot y$ (projection substitution),
2. $t_3 \leftarrow (x\ S_1)$ (for direct long head-normalisation of t_1 before passing it to SIMPL), and
3. $t_1 \leftarrow t_2$ (substituting equal for equal).

Remember that unknowns, abstractions, and potential redexes are all represented by muterms. So, they are reversibly mutable.

The conclusion is that much more substitutions than the so-called *solution substitutions* are done. The supplementary substitutions contribute to saving unification and reduction time, and to saving memory.

4.1.5 Proof-search

Prolog control The representation of the search-stack controlling the search process uses MALI's term-stack. It is considered as a *failure continuation*. Specialised commands are defined for manipulating the failure continuation. The representation of the proof-stack controlling the development of the proof tree also uses MALI. It is mapped on compound terms. It is considered as a *success continuation*, and other commands are defined for manipulating it.

Since the term-stack and compound-terms are regular MALI's terms, we have a uniform representation of λProlog terms and control. This makes continuation capture (of both kinds) trivial. It appears that implementing the Prolog cut merely requires to capture the failure continuation when entering a clause (a reification) and reinstalling it (a reflection) when executing the cut predicate. All this comes for free by using MALI.

Given program

```
in (f a).  in (f b).
trans (f X) (g X).
out X :- ...
:- in I, trans I O, /*1*/ !, out O.
```

when label `/*1*/` is reached, the resolution state can be represented in MALIv06 like

```
success_continuation =
        cut_goal@(tu S_GOAL 3 (at S_SYMB cut)
                     eos@(le S_CHPT
                             (c2 S_ROOT (at S_INT 2)
                                 (tu S_GOAL 2 (at S_SYMB end_of_search) (c0 S_NIL))
                          _)
        out_goal@(tu S_GOAL 3 (at S_SYMB out)
                     O@ga@(tu S_APPL 2 (at S_SYMB g) (at S_SYMB a))
             eop@(tu S_GOAL 2 (at S_SYMB end_of_proof) (c0 S_NIL))))


failure_continuation =
        in2@(le S_CHPT
                (c2 S_ROOT (at S_INT 2)
                    (tu S_GOAL 3 (at S_SYMB in) I@(mc2 S_UNK type (at S_SIG 0))
                    (tu S_GOAL 4 (at S_SYMB trans) I O@(mc2 S_UNK type' (at S_SIG 0))
                    cut_goal)))
             eos)
```

Label `O` occurs in `success_continuation` and `failure_continuation` accompanied with different terms. Terms in `success_continuation` differ by a substitution from terms with same labels in `failure_continuation`, but they share the same representation anyway. We leave unspecified the types `type` and `type'` of unknowns `I` and `O`. Note that the argument of goal `!` is a substack of the search-stack. Binary constructs of sort `S_ROOT` represent the roots of the choice-points. They contain a clause number and a success continuation. After goal `!` is executed, the state is

```
success_continuation = out_goal
failure_continuation = eos
```

In real-life, the first goal of a success continuation is dispatched into several registers. This saves "consing" and "deconsing" the continuation.

Universally quantified goals They are implemented as we have said about universal variables. The current nesting level is in fact a *signature continuation.* It has the same search-dynamism as the success continuation. This means that it is saved (i.e. pushed on MALI's term-stack) and restored (i.e. popped from MALI's term-stack) with the success continuation.

Implication goals Implication is the other new construct of λProlog that enforces checking scoping conditions. The premise of an implication goal must be added to the program for the length of the proof of its conclusion.

Every premise is compiled as a clause whose logical variables are the proper logical variables of the premise, plus the logical variables of the nesting clause that occur in the premise. Premises are *activated* when their implication goals are executed. The scope of premises is controlled by a *program continuation* [7] that is implemented as MALI's terms, and has the same search-dynamism as the success continuation and the signature continuation. The program continuation is made of closures that enrich every active premise with a context corresponding to the logical variables of the nesting clause that occur in the premise.

Predicates that can be extended by implication are declared *dynamic* so that not every predicate pays for implication. When a goal of a dynamic predicate is executed, one first searches the program continuation for matching premises.

This scheme is similar to what Jayaraman and Nadathur propose [19]. The only difference is that there is only one thing to say about the interferences with backtracking: it is automatically done by MALI.

4.1.6 A memory policy

The choice of a memory policy was left undefined at the level of MALI. It is still too soon to wire it at the level of the SIM because the same machine will be used in λProlog applications with totally different memory requirements (any combination of consumption rate and instantaneous working space). Since generated applications are portable, the same machine will also be used in different configurations of host systems (any combination of CPU speed and sizes of main memory and secondary memory).

We designed a memory policy which is both parameterisable and adaptative. The supplies given to MALI, the part it actually uses, and other parameters are continuously monitored, and evolution parameters are changed automatically. However, this may not be flexible enough and every executable file resulting from the compilation of a Prolog/Mali program accepts conventional arguments for configuring the memory policy to the users's will.

5 A compilation scheme

λProlog programs are translated into C programs which serve as a glue for putting together sequences of SIM commands. The use of C is purely incidental, but its availability and portability are good points. The C program is compiled with the regular C compiler/linker, producing an executable file for the host system. The generated C program is responsible for realising the standard interface (call/return conventions, input/output ports) with the host system.

The commands of the specialised intermediate machine are assembled so that when the generated program is executed, it has the intended proof-search behaviour. Many arrangements are possible for producing the intended behaviour. Compiling becomes really valuable when special source patterns exist for determining efficient arrangements. Efficient arrangements are in fact spe-

cialisations of a general execution scheme. We list the patterns our compiler currently recognises and the associated specialisations and savings.

5.1 Special static patterns

5.1.1 Forgotten types

We have seen that types must be represented to some extent at run-time. A naive solution would be to represent the types of every term and subterm. The important pattern that improves the representation of types is the occurrence of forgotten types in type declarations. They indicate the only places in which types need to be represented for checking the well-typing of unification problems.

Furthermore, the type checking/inference done at compile-time indicates which types are identical and can share representation.

Type declarations are translated into type reconstruction functions (also coded in C).

5.1.2 Combinators

β-reduction requires duplicating left members of redexes. It is easy to see that combinators need not be duplicated and that their representation can be shared.

Since substitution values are always combinators, all instances of combinators of the source program are combinators. So, it is worth recognising them at compile-time. Our experiments show that it is a very important pattern, and that using it properly changes the complexity of programs [9].

5.1.3 First-order applications and constants

The general unification procedure of λProlog is Huet's procedure augmented with dynamic type checking. However, first-order terms deserve a more direct unification procedure. So, these patterns are compiled rather classically. The representation of first-order applications is chosen to be easily recognised so that, at run-time, unification and β-reduction are improved.

5.1.4 $\beta\eta$-normalisation

Source clauses are $\beta\eta$-normalised before generation. This provides a macro-like feature which may improve the programming style. Furthermore, first-order applications are put in η-long form. This makes dynamic long head-normalisation less necessary.

η-expansion must be done carefully so that it does not create artificially large β-redexes. So, abstractions that are created by η-expansions are tagged, and β-redexes built with them are reduced using equality $(\lambda_\eta x.(E\ x)\ F) =_\beta (E\ F)$. New sorts are required for representing the terms of the cross-product (combinator/non-combinator)×(eta-expanded/non-eta-expanded). Again, it is a very important pattern that changes the complexity of programs [8].

5.1.5 A weak substitute for clause indexing

Clause indexing is the exploitation of the clause heads contents for computing more direct clause selection procedures. It is not yet implemented in Prolog/Mali.

Usually, when control enters a clause that is not the last clause of a predicate, a choice-point is created (or an already existing choice-point is updated). It can be a waste of time and memory if a succession of choice-point creations and choice-point consumptions is used to select a clause in a predicate. Clause indexing helps selecting more directly the proper clause.

The lack of clause indexing is somewhat compensated by delayed creation of choice-points. Delayed creation of choice-points amounts to indicating that a choice-point is to be created instead of creating it. The creation must be resumed as soon as a logical variable is substituted, or when unification succeeds (if no logical variable is substituted). If a failure occurs while the choice-point creation is still delayed, failure is merely implemented as a jump.

More interestingly, substitutions of a head-normal-form to a non-normal form do not count as substitutions of logical variables. So, they do not trigger the choice-point creation. The neat effect is that a goal argument will be reduced only once for all the attempts at unifying a clause head, whereas if the choice-point were created as soon as ordered then the goal argument would be reduced for every unification attempt, and unreduced at every backtrack. For instance, in

```
test 0 :- do_something.
test 1 :- do_something_else.
query  :- N = s\z\ (s(s(s(s(s(s z)))))), M = 1, test (N x\x M).
```

redex `(s\z\(s(s(s(s(s(s z)))))) x\x 1)` is reduced only once instead of twice. Note that the brute force solution consisting in reducing a goal before unifying

1. kills lazyness,
2. and does not eliminate the need for normalising during unification because substitutions might build redexes.

So, delayed creation of choice-point gives a partial solution to a critical problem that appears every time normalisation of terms or awakening of constraints are possible.

5.2 The translation

λProlog programs are translated into C on a predicate-to-function basis. Every predicate is implemented as a function of the continuations (success, signature, program, and failure) that returns new values for the continuations.

The functions never call each other; recursion is taken into account by the success continuation. Functions are called by, and return to, a *motor*, which can be considered the last remnant of an interpreter. Some static patterns, such as left-recursion, allow to avoid going through the motor.

As we have seen in section 4.1.1, type schemes are translated into type reconstruction functions. Furthermore, every constant (predicate and function constants, and type constructors) is translated into a C structure containing their external representation, their arity, their predicate function or type reconstruction function, if needed, and any useful information.

6 Conclusions

6.1 Prolog/Mali

The Prolog/Mali compiler is written in λProlog and the run-time libraries are written in λProlog and in C. The Prolog/Mali system has been completely bootstrapped. It implements all the core of λProlog plus various extensions. One of the most notable extensions is the continuation capture capability. It is used for implementing the cut and a catch/throw escape system.

Prolog/Mali is freely available, and used in several research teams in domains such as automated theorem proving, automated learning, and meta-programming. Some of the benchmarks used for comparing Prolog/Mali with other implementations come from these teams.

6.2 Comparisons with other works

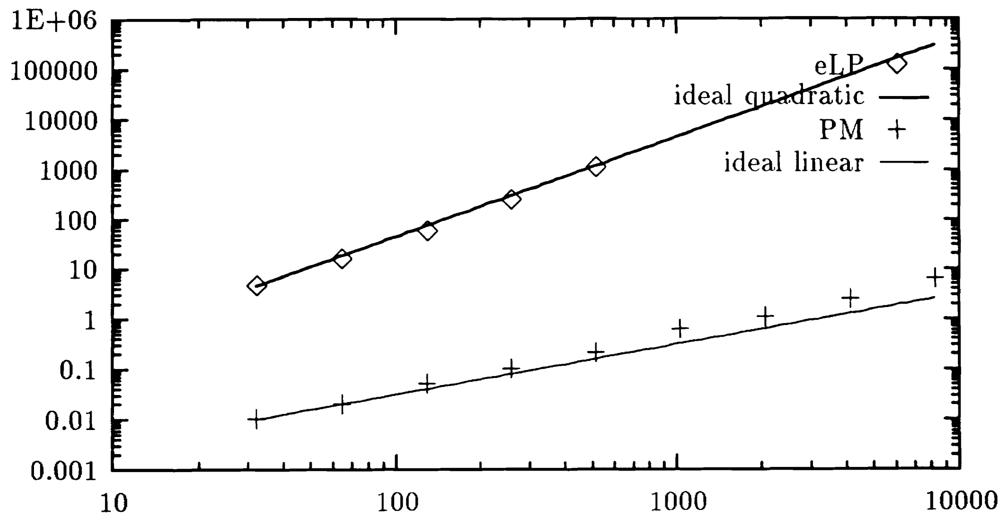


Figure 1: Comparison of time complexities when reversing a function-list (list-length × run-times in seconds, log-log scale)

It has not been possible to compare our system with the other most recent attempts for implementing λProlog (Nadathur, Jayaraman, Felty), because of the lack of availability of complete systems. However, papers and technical reports by Nadathur and Jayaraman [33, 20, 19] show that their approach and ours are somewhat different and difficult to compare on the paper. In few words, they choose to base their design on a WAM augmented for handling λProlog's specifics.

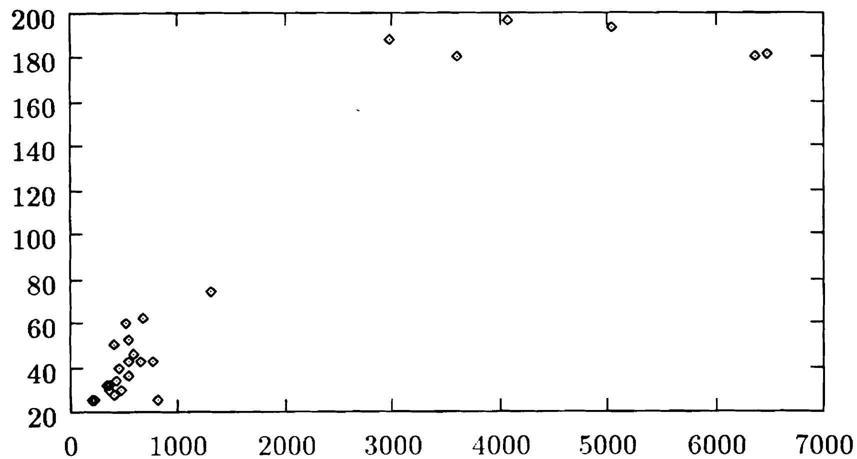


Figure 2: Comparison of run-times when executing a tactical theorem prover (Prolog/Mali run-times in seconds × speed-ratio)

They represent λ-terms and reduction in an environment-based fashion. Note that the differences may be blurred by optimisations that apply techniques from one paradigm for improving the other.

A technical report by Kwon, Nadathur and Wilson [22] proposes a handling of types at run-time which is similar to ours, except that forgotten types are not the only types represented in constants. Note that their basic technical choice, and Jayaraman's, is to extend a structure-sharing implementation of the WAM: it also applies to the representation of types.

The only λProlog system with which we have made extensive comparisons is eLP. It is already an "old" system. eLP is an interpreted system written in Lisp. The fact that it is interpreted could have explained a constant speed factor between eLP and Prolog/Mali. However, what is observed is a difference in complexity that interpretation costs cannot explain alone. We compared Prolog/Mali and eLP in a black-box mode, knowing nothing of the implementation of eLP. The comparison has been done using special purpose programs for exhibiting qualitative differences, and also using regular programs from λProlog users.

The memory management improvement over eLP is dramatic for any kind of program. It is also better than many implementations of standard Prolog. The Lisp system that supports eLP has its own memory management, which might be efficient as far as Lisp evaluation is concerned. But it does not know about logic programming usefulness logic, and does nothing when early reset and muterm shunting are in order. It is a definitely bad idea to leave a non logic programming system in charge of logic programming memory management. Note that this does not forbid implementing logic programming in a foreign language; the only thing is that logic programming memory management has to be redone in that language.

Special purpose programs show an arbitrary speed-up of Prolog/Mali over eLP's. The com-

plexity of both unification and reduction is higher in eLP. We believe that the systematic sharing and folding of representations, and the detection of combinators play a critical part in the better complexity of Prolog/Mali. Delaying the creation of choice-points also improves the complexity of search. Figure 1 shows the behaviours of eLP and Prolog/Mali when executing the program that naively reverses a function list. Times are given in seconds as a function of the length of a list. Scales are logarithmic on both axes. Continuous lines correspond to the ideal linear or quadratic case. The slopes of the lines, 1 and 2, indicate a linear complexity for the first and a quadratic complexity for the second.

Regular programs (mainly a demonstrator with tacticals, and a demonstrator with a learning component) show a speed-up between 25 and 250. Interestingly enough, for a given program, the speed-up grows with the time required for executing a query. This shows that eLP does not scale up very well. Figure 2 shows the speed-up of Prolog/Mali over eLP for a set of small theorem proving problems. Every point correspond to a particular problem. Execution times with Prolog/Mali are on the X-axis and the speed-ups (Prolog/Mali on eLP) are given on the Y-axis.

Finally, we compared Prolog/Mali with modern (fast) implementations of standard Prolog. When using regular programs (mainly an early version of our compiler), Prolog/Mali is less than 10 times slower than Prolog (≈ 5 on the average). Special purpose programs could show arbitrary differences (e.g. we have not yet implemented clause indexing in Prolog/Mali). This comparison is a little bit unfair for Prolog/Mali, and for λProlog in general, because it executes the first-order Horn clauses fragment of λProlog with a higher-order hereditary Harrop formulas technology. When what the user requires is exclusive to λProlog, the standard Prolog programmer has to implement it at the Prolog level; it is certainly less efficient, and less safe too, than what a λProlog system offers.

6.3 Further work

Although our implementation of λProlog enjoys nice complexity properties, and its performances are encouraging, it is rather slow when it is compared with the current state of the art for standard Prolog. In its present state the control of search is compiled but unification of higher-order terms is not and there is no clause indexing. Our current implementation task is to devise a compilation scheme for unification and indexing so as to bring the performance level of the standard part closer to the current state of the art.

To improve performances, more static analysis ought to be performed. For instance, it is important to detect when the full mechanism of Huet's unification is not needed. The L_λ [25] fragment of λProlog has a unitary and decidable unification theory. Belonging to L_λ is easy to test at run time but it could be more efficient to detect that some predicate or some argument will always be in L_λ. Note that the L_λ property generalises every pattern that the TRIV procedure currently recognises.

Last observation is that the type system deserves further study. It should be studied for itself because it is not flexible enough. It should also be studied for its interaction with compilation (indexing and projection). By flexibility, we do not mean permissivity, but only the ability to deal with complex situations. The lack of flexibility is in fact nothing special to λProlog, it can already be observed in trying to type built-in predicates `read` and `name` in Typed Prolog [31, 23]. It is only

more noticeable in λProlog than in Typed Prolog because types are mandatory whereas they are only a bonus in Typed Prolog. Predicates `read` and `name` can only be simply typed like

```
type  read
        _ -> o.
type  name
        (list int) -> _ -> o.
```

These predicates with these typings are not definitionally generic, and the arguments with the anonymous types cannot be used soundly in any specific context because their types are related with nothing. It seems that we need higher-order types such as

```
type  read
        'PI' A\ (A -> o).
type  name
        (list int) -> 'PI' A\ (A -> o).
```

6.4 Remarks on focusing on memory management

Our main implementation concern has been memory management. We always tried to have memory management problems solved before time efficiency problems. This is reflected in the software architecture of Prolog/Mali, in which the kernel (MALI) knows almost everything about memory management but nothing on the procedures that will be used, the specialised abstract machine knows less about memory management, and a little bit more about the procedures, and the generated code knowns about the procedures (it is part of them) but is really naïve as far as memory management is concerned.

However, a reasonable time efficiency has been achieved, and still more can be gained with further efforts.

This architecture can be used for implementing many other kinds of logic programming systems. It cannot compete for implementing standard Prolog systems because very efficient and specialised techniques have already been designed. It is perfectly fit as soon as complex data-structure and control are in order. An implementation redoing a specialised version of Mali's memory management from scratch could always be faster but will certainly be much more complex.

Acknowledgements

We thank Yves Bekkers, Serge Le Huitouze, Barbara Oliver, and the reviewers from the Lambda Prolog Workshop committee for their reading and commenting of earlier versions of this paper.

References

[1] M. Abadi, L. Cardelli, B.C. Pierce, and G.D. Plotkin. *Dynamic Typing in a Statically Typed Language.* Technical Report, DEC Systems Research Center, 1989.

[2] H. Aït-Kaci. *The WAM: A (Real) Tutorial.* Technical Report 5, DEC Paris Research Laboratory, 1990.

[3] H. Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* MIT Press, 1991. Revised after [2].

[4] J.-M. Andreoli and R. Pareschi. Linear objects: logical processes with built-in inheritance. In D.H.D. Warren and P. Szeredi, editors, *7th Int. Conf. Logic Programming*, MIT Press, Jerusalem, Israel, 1990.

[5] H. Barendregt. Introduction to generalized type systems. *J. Functional Programming*, 1(2):125–154, 1991.

[6] Y. Bekkers, B. Canet, O. Ridoux, and L. Ungaro. MALI: a memory with a real-time garbage collector for implementing logic programming languages. In *3rd Symp. Logic Programming*, IEEE, Salt Lake City, UT, USA, 1986.

[7] P. Brisset. *Compilation de λProlog.* Thèse, Université de Rennes I, 1992.

[8] P. Brisset and O. Ridoux. *The Compilation of λProlog and its execution with MALI.* Publication Interne, IRISA, 1992. To appear.

[9] P. Brisset and O. Ridoux. Naïve reverse can be linear. In K. Furukawa, editor, *8th Int. Conf. Logic Programming*, pages 857–870, MIT Press, Paris, France, 1991.

[10] C.M. Elliott. Higher-order unification with dependent function types. In N. Derschowitz, editor, *3rd Int. Conf. Rewriting Techniques and Applications*, pages 121–136, Springer-Verlag, 1989. LNCS 355.

[11] C.M. Elliott and F. Pfenning. *A Semi-Functional Implementation of a Higher-Order Logic Programming Language.* Ergo Report ERGO–89–080, School of Computer Science, Carnegie Mellon University, 1989.

[12] A. Felty. *Specifying and Implementing Theorem Provers in a Higher-Order Logic Programming Language.* PhD Dissertation, Dept. of Computer and Information Science, University of Pennsylvania, 1989.

[13] A. Felty and D.A. Miller. *Encoding a Dependent-Type λ-Calculus in a Logic Programming Language.* Rapport de Recherche 1259, Inria, 1990.

[14] A. Felty and D.A. Miller. Specifying theorem provers in a higher-order logic programming language. In E. Lusk and R. Overbeek, editors, *CADE-88*, pages 61–80, Springer-Verlag, Berlin, FRG, 1988. LNCS 310.

[15] M. Hanus. Horn clause programs with polymorphic types: semantics and resolution. *Theoretical Computer Science*, (89):63–106, 1991. Previously in [16].

[16] M. Hanus. Horn clause programs with polymorphic types: semantics and resolution. In *TAPSOFT'89*, pages 225–240, Springer-Verlag, Barcelona, Spain, 1989. LNCS 352.

[17] J.S. Hodas and D.A. Miller. Logic programming in a fragment of intuitionistic linear logic. In G. Kahn, editor, *Symp. Logic in Computer Science*, pages 32–42, Amsterdam, The Netherlands, 1991.

[18] G. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, (1):27–57, 1975.

[19] B. Jayaraman and G. Nadathur. Implementation techniques for scoping constructs in logic programming. In K. Furukawa, editor, *8th Int. Conf. Logic Programming*, pages 871–886, MIT Press, Paris, France, 1991.

[20] B. Jayaraman and G. Nadathur. Implementing λProlog: a progress report. In *2nd NACLP Workshop on Logic Programming Architectures and Implementations*, MIT Press, 1990.

[21] A.J. Kfoury, J. Tiuryn, and P. Urzyczyn. *The Undecidability of the Semi-Unification Problem.* Technical Report BUCS 89-010, Boston University, 1989.

[22] Keehang Kwon, G. Nadathur, and D.S. Wilson. *Implementing Logic Programming Languages with Polymorphic Typing.* Technical Report CS–1991–39, Dept. of Computer Science, Duke University, 1991.

[23] T.K. Lakshman and U.S. Reddy. Typed Prolog: a semantic reconstruction of the Mycroft-O'Keefe type system. In *Int. Logic Programming Symp.*, pages 202–217, 1991.

[24] S. Le Huitouze. A new data structure for implementing extensions to Prolog. In P. Deransart and J. Małuszyński, editors, *Int. Work. Programming Languages Implementation and Logic Programming*, Springer-Verlag, 1990. LNCS 456.

[25] D.A. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. In P. Schroeder-Heister, editor, *Int. Workshop on Extensions of Logic Programming*, Springer-Verlag, New York, Tübingen, FRG, 1989. LNAI 475.

[26] D.A. Miller. A logical analysis of modules in logic programming. *J. Logic Programming*, 6(1–2):79–108, 1989.

[27] D.A. Miller. A theory of modules for logic programming. In *Symp. Logic Programming*, pages 106–115, Salt Lake City, UT, USA, 1986.

[28] D.A. Miller and G. Nadathur. Higher-order logic programming. In E. Shapiro, editor, *3rd Int. Conf. Logic Programming*, pages 448–462, Springer-Verlag, London, UK, 1986. LNCS 225.

[29] D.A. Miller and G. Nadathur. A logic programming approach to manipulating formulas and programs. In *IEEE Symp. Logic Programming*, pages 379–388, San Francisco, CA, USA, 1987.

[30] D.A. Miller, G. Nadathur, and A. Scedrov. Hereditary Harrop formulas and uniform proof systems. In *2nd Symp. Logic in Computer Science*, pages 98–105, Ithaca, New York, USA, 1987.

[31] A. Mycroft and R.A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, (23):295–307, 1984.

[32] G. Nadathur. *A Higher-Order Logic as the Basis for Logic Programming.* Ph.D. Thesis, University of Pennsylvania, 1987.

[33] G. Nadathur and B. Jayaraman. Towards a WAM model for λProlog. In E.L. Lusk and R.A. Overbeek, editors, *1st North American Conf. Logic Programming*, pages 1180–1198, MIT Press, 1989.

[34] G. Nadathur and D.S. Wilson. Representation of lambda terms suitable for operations on their intensions. In *ACM Conf. Lisp and Functional Programming*, pages 341–348, ACM Press, Nice, France, 1990.

[35] T. Nipkow. *Higher-Order Unification, Polymorphism, and Subsorts.* Technical Report 210, University of Cambridge, Computer Laboratory, 1990.

[36] F. Pfenning. Logic programming in the LF logical framework. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[37] F. Pfenning. Unification and anti-unification in the calculus of constructions. In *Symp. Logic in Computer Science*, pages 74–85, 1991.

[38] O. Ridoux. *MALIv06: Tutorial and Reference Manual.* Publication Interne 611, IRISA, 1991.

[39] D.H.D. Warren. *An Abstract Prolog Instruction Set.* Technical Note 309, SRI International, 1983.

Higher-Order Substitutions (Preliminary Results)

D. Duggan [1]
Department of Computer Science
University of Waterloo
Waterloo, Ontario N2L 3G1, Canada
dduggan@plg.uwaterloo.ca

1 Abstract

Languages such as λ-Prolog and Elf advocate an approach to program manipulation based on higher order abstract syntax, with substitution built in to the language evaluator. Recently substitution has received fresh attention with concrete versions of the λ-calculus where substitutions are made explicit as terms in the language. In this paper we show how explicit substitutions may be introduced into a language for manipulating higher order abstract syntax. The implementation of full substitution in the evaluator may be avoided by using a metalanguage which supports a generalization of Miller's *patterns*. We briefly comment on the motivation for such an approach to substitutions.

2 Introduction

"I don't really like deBruijn numbers myself." N. G. deBruijn.

The $\lambda\sigma$-calculus [1] has recently been proposed as a formalism for reasoning about implementations of the λ-calculus. This formalism is based on a concrete formulation of the λ-calculus where variables are replaced by deBruijn numbers [5], and where substitutions are made explicit in the (two-sorted) term language. Applications of this calculus include the derivation of a Krivine-like abstract machine and a type-checker for the second-order λ-calculus. A similar system (ΛCCL) has been independently developed by Field [9], who has also developed a labelled version of his system to reason about optimality. Another similar system has been proposed by Nadathur and Wilson as a foundation for implementations of λ-Prolog [17]..

In this paper we propose a similar system which incorporates explicit substitutions into the λ-calculus. However in contrast to the first-order approaches mentioned above, our system is based on *higher-order abstract syntax* [16, 8]: rather than representing variables concretely as deBruijn numbers, we represent them instead as variables in the metalanguage, with variable binding in the object language λ-calculus represented by λ-abstraction in the metalanguage. We formulate typing and equality rules for this calculus where applications of "free" function variables employ an extension of a restriction discovered by Miller [15] (see also [18, 19]). Our extension enjoys the same pleasing properties of decidability and most general unifiers that Miller's *patterns* ensure (the

[1] Supported in part by the NSERC grant OGP0105568.

details are worked out in a companion paper [7]). We make essential use of our generalization of patterns to product types in what follows.

Since our interest is in specifying and implementing type-checkers for languages with higher-order type systems (e.g. Quest [3], Pebble [2]), we present a type-checker based on explicit higher-order substitutions. Viewed as a (determinate) logic program, this type-checker can be implemented directly in a language which supports products, extended patterns and polymorphic (non-uniformly parameterized) data types.

With the reader's indulgence, we use variations of the same λ-calculus for both metalanguage and object language in this paper. The core λ-calculus is Luo's Extended Calculus of Constructions [14], a system with predicative general products (dependent function types) and general sums (dependent sum types), a cumulative hierarchy of type universes, and impredicative logical quantifiers. We have designed an L_λ-like logic programming language, based on placing syntactic restrictions on this calculus, which ensure decidable unification and a complete operational semantics relative to a realizability semantics. The type-checking algorithm provided in Section 6 is implementable with minor modifications in this metalanguage. The object language is Luo's system restricted to general products and type universes, where the main issues arise. Thus the type-checker we develop as a metalanguage program may be considered as a type-checker for the metalanguage.

Regarding the usefulness of this approach, we hope that it will aid in the development of automated reasoning and programming environment tools based on higher-order abstract syntax. For example it may serve as the basis for providing explicit substitutions as "classes" in a metalanguage with an appropriate notion of "inheritance." Finally we conjecture that further enrichments of the metalanguage may strengthen the power of the formalism for reasoning about reduction strategies; for example the addition of linear connectives may enable us to reason in the metalanguage about sharing [13, 10], a deficiency with the $\lambda\sigma$-calculus [4, 9].

3 Luo's Extended Calculus of Constructions

The core λ-calculus we will be using for both metalanguage and object language is Luo's Extended Calculus of Constructions. We will not concern ourselves too much with the structure of the metalanguage (details are provided elsewhere [6]). The salient features are:

1. a special constant **Type** representing the "kind" of all types;

2. a dependent function type $\Pi x : A \cdot B$, including quantification over types (terms of kind **Type**);

3. λ-abstraction for representing object language terms with variable binding, with $\lambda x : A \cdot M \in \Pi x : A \cdot B$;

4. application $M(N)$ for $M \in \Pi x : A \cdot B$. $N \in A$; and

5. products (pairs), with product type $A \times B$ and left and right projections $\pi_1(M)$ and $\pi_2(M)$.

For readability we will adopt the abbreviation $M(N_1; \ldots; N_n)$ for $(\ldots(M(N_1))\ldots N_n)$ (more traditionally written as $(M\ N_1 \ldots N_n)$). Also $[n]$ will denote the set $\{1, \ldots, n\}$. We will adopt the traditional abbreviation that $A \rightarrow B$ denotes the function type $\Pi x : A \cdot B$ where $x \notin \mathtt{FV}(B)$.

To ensure decidable unification, Miller has proposed restricting applications of a "free" function variable F to have the form $F(x_1, \ldots, x_n)$, where the x_i's are λ-bound and distinct [15] (see also [18, 19]). With the introduction of product types, this restriction can be generalized to allow applications of the form

$$F(p_1(x_1), \ldots, p_n(x_n))$$

where each p_i is a sequence of projections applied to a λ-bound variable x_i (i.e. $p_i(x_i) \equiv \pi_{i_1}(\ldots(\pi_{i_{m_i}}(x_i))\ldots))$, and where moreover if $x_i = x_j, i \neq j$, then neither p_i nor p_j are prefixes of each other. Note in particular that this allows repeated occurences of a λ-bound variable in a pattern. Decidability of unification and most general unifiers are maintained with these generalized patterns [7]. We make essential use of these generalized patterns in composing higher order substitutions, discussed in the next section.

The foundations for this metalanguage lie in Luo's Extended Calculus of Constructions [14]. The major difference between ECC and the metalanguage just described is that the former explicitly stratifies types into a cumulative hierarchy of type universes. For terms of the metalanguage we will leave this stratification implicit [12]. However we make this stratification explicit when we take (a subset of) Luo's ECC as the object language. We will provide a slightly non-traditional presentation of a subsystem of ECC (restricted to dependent function types and type universes) using higher order abstract syntax. This will serve to demonstrate the use of higher-order substitutions both for implementing β-reduction and for type-checking with dependent types[2]. A representation for terms of our ECC subset is given by the following metalanguage signature:

```
 Term  :  type
 type  :  Nat → Term
   pi  :  Term → (Term → Term) → Term
  abs  :  Term → (Term → Term) → Term
apply  :  Term → Term → Term
```

Terms in the object language have the form:

$\mathtt{type}(i)$, $\mathtt{pi}(A; B)$, $\mathtt{abs}(A; M)$, $\mathtt{apply}(M, N)$

representing respectively (the name of) a type universe, the dependent function type, λ-abstraction and application.

Figure 1 in the Appendix gives the typing rules for the object language. To keep the number of rules to a minimum we present the system using equality judgements $\Gamma \rhd M = N \in A$, with the abbreviation:

$$\Gamma \rhd M \in A \stackrel{\text{def}}{=} \Gamma \rhd M = M \in A$$

[2]Although we could have used e.g. the second order λ-calculus as a possibly more familiar example for type-checking, our presentation is shortened using ECC because of the common structure for terms and types.

We will use judgements of the following forms:

$$\begin{array}{llll} \text{Environments} & \Gamma & ::= & \mathsf{nil} \mid \Gamma, x : A \\ \text{Judgements} & \mathcal{J} & ::= & \Gamma\ \mathbf{env} \mid \Gamma \rhd M = N \in A \mid \Gamma \rhd M \preceq N \in A \end{array}$$

Note that the following rule is derivable from CUM:

$$\text{EQ}\ \frac{\Gamma \rhd M = M' \in A \quad \Gamma \rhd A' \in \mathtt{type}(i) \quad \Gamma \rhd A = A' \in \mathtt{type}(i)}{\Gamma \rhd M = M' \in A'}$$

We will refer to the system in Figure 1 as $\Lambda\beta$.

The equality rules, oriented as a rewrite system, are obviously confluent and Church-Rosser[3]. Denote the judgement that M rewrites to N by $\Gamma \rhd M \rightsquigarrow_\beta N \in A$, and let $\Gamma \vdash_{ECC} \mathcal{F}$ denote derivability of the judgement $\Gamma \rhd \mathcal{F}$ using the rules of $\Lambda\beta$. Then Luo has verified the following properties for ECC:

Proposition 3.1 *The following properties are true of ECC [14]:*

Church-Rosser *If* $\Gamma \rhd N_1 = N_2 \in A$, $\Gamma \rhd N_1 \in A$ *and* $\Gamma \rhd N_2 \in A$, *then there is some* M *such that* $\Gamma \rhd N_1 \rightsquigarrow_\beta M \in A$ *and* $\Gamma \rhd N_2 \rightsquigarrow_\beta M \in A$.

Subject Reduction *If* $\Gamma \rhd M \in A$ *and* $\Gamma \rhd M \rightsquigarrow_\beta N \in A$, *then* $\Gamma \rhd N \in A$.

Strong Normalization *If* $\Gamma \rhd M \in A$ *then* M *is strongly normalizable.*

Decidable Type-Checking *Type checking, convertibility and cumulativity are decidable.*

Minimal Types *Any well-typed term* M *of ECC has a minimal type* A *such that (1)* $\Gamma \rhd M \in A$ *and (2) for any* A' *such that* $\Gamma \rhd M \in A'$, *we have* $\Gamma \rhd A \preceq A \in \mathtt{type}(i)$ *for some* $i \in \omega$.

In the next section it will be useful to consider reduction on untyped terms of $\Lambda\beta$; we will denote this by $M \rightarrow_\beta N$. Note that the Church-Rosser property still holds for untyped reduction due to the absence of critical pairs.

4 ECC With Substitutions

The formulation of the $\Lambda\beta$ object language in the previous section relied in several places on the use of β-reduction to implement substitution. In this section we remove this reliance on β-reduction in the metalanguage by making substitutions explicit in the object language. For brevity we refer to the resulting system as $\Lambda B\sigma$.

[3] We have omitted the equality rule:

$$\text{ETA}\ \frac{\Gamma \rhd M \in \mathrm{pi}(A; B)}{\Gamma \rhd \mathtt{abs}(A; \lambda x \cdot \mathtt{apply}(M; x)) = M \in \Pi(A; B)}$$

since confluence fails with a naive equality relation (because of cumulativity).

We introduce a new type constructor `Subst` into the object language for substitutions. In our system substitutions will be trees of (value,type) pairs (rather than lists as in the $\lambda\sigma$-calculus and ΛACCL). Thus the (metalanguage) type of a substitution is parameterized by a product type reflecting the structure of the substitution. Note that we are making non-trivial use of both product types and polymorphism in the definition of substitutions. The additions to the object language signature of the previous section are[4]:

$$
\begin{array}{lcl}
\texttt{Subst} & : & \texttt{type} \rightarrow \texttt{type} \\
\texttt{clos}' & : & \Pi \texttt{S} : \texttt{type} \cdot (\texttt{S} \rightarrow \texttt{Term}) \rightarrow \texttt{Subst(S)} \rightarrow \texttt{Term} \\
\texttt{map}'' & : & \Pi \texttt{S}_1 : \texttt{type} \cdot \Pi \texttt{S}_2 : \texttt{type} \cdot (\texttt{S}_1 \rightarrow \texttt{Subst}(\texttt{S}_2)) \rightarrow \texttt{Subst}(\texttt{S}_1) \rightarrow \texttt{Subst}(\texttt{S}_2) \\
[_,_] & : & \texttt{Term} \rightarrow \texttt{Term} \rightarrow \texttt{Subst(Term)} \\
_ \circ _'' & : & \Pi \texttt{S}_1 : \texttt{type} \cdot \Pi \texttt{S}_2 : \texttt{type} \cdot \texttt{Subst}(\texttt{S}_1) \rightarrow \texttt{Subst}(\texttt{S}_2) \rightarrow \texttt{Subst}(\texttt{S}_1 \times \texttt{S}_2)
\end{array}
$$

Here the `subst` term constructor represents the application of a substitution to a term. Basic substitutions are built using the $[_,_]$ constructor. Thus whereas in $\Lambda\beta$ we had

$$
\textsc{Beta}\quad \frac{\Gamma, x : A \ \triangleright M(x) \in B(x) \qquad \Gamma \triangleright N \in A}{\Gamma \ \triangleright \texttt{apply}(\texttt{abs}(A; M); N) = M(N) \in B(N)}
$$

in $\Lambda B\sigma$ the rule is formulated as

$$
\textsc{Beta}\quad \frac{\Gamma, x : A \ \triangleright M(x) \in B(x) \qquad \Gamma \triangleright N \in A}{\Gamma \ \triangleright \texttt{apply}(\texttt{abs}(A; M); N) = \texttt{clos}(M; [N, A]) \in \texttt{clos}(B; [N, A])}
$$

These `clos` terms are similar to the *higher-order closures* introduced by Hannan and Miller [11]. For this approach to be useful we must be able to maintain these closures in the form $\texttt{clos}(\lambda x \cdot \texttt{t}(M_1(x); \ldots; M_n(x)); s)$ where `t` is the outermost term constructor (not `clos`). Therefore we have the following rule for composing substitutions:

$$
\textsc{SubstSubst}\quad \frac{\Gamma \ \triangleright \texttt{clos}(\lambda x \cdot M(\pi_1(x); \pi_2(x)); s_2 \circ \texttt{map}(s_1; s_2)) \in C}{\Gamma \ \triangleright \texttt{clos}(\lambda x \cdot \texttt{clos}(M(x); s_1(x)); s_2) \sim \texttt{clos}(\lambda x \cdot M(\pi_1(x); \pi_2(x)); s_2 \circ \texttt{map}(s_1; s_2)) \in C}
$$

This rule makes use of the two other constructors for substitutions: $_ \circ _$ forms the composition of two substitutions, while `map` applies a substitution to another substitution (In $\lambda\sigma$-calculus and ΛACCL, these constructors are combined into a single composition operator, with a reduction rule mapping the second substitution over the first). For the purposes of higher-order abstract syntax, the crucial point is that whereas the original term M has two free variables being substituted for by two separate substitutions, the resulting term has one free variable being substituted for by a single composite substitution, with the previous free variables specialized to projections out of this composite substitution. The rules for applying a substitution ("projecting out of an environment") then rely on matching against the projections inserted by the composition rule:

$$
\textsc{SubstL}\quad \frac{\Gamma \ \triangleright s_2 \in \texttt{Subst}(S) \qquad \Gamma \ \triangleright \texttt{clos}(M; s_1) \in B}{\Gamma \ \triangleright \texttt{clos}(\lambda x \cdot M(\pi_1(x)); s_1 \circ s_2) \sim \texttt{clos}(M; s_1) \in B}
$$

$$
\textsc{SubstR}\quad \frac{\Gamma \ \triangleright s_1 \in \texttt{Subst}(S) \qquad \Gamma \ \triangleright \texttt{clos}(M; s_2) \in B}{\Gamma \ \triangleright \texttt{clos}(\lambda x \cdot M(\pi_2(x)); s_1 \circ s_2) \sim \texttt{clos}(M; s_2) \in B}
$$

[4]The quoting annotation $'$ is borrowed from LEAP [20], and signifies inference of an implicit (type) parameter based on the types of the remaining arguments.

Figure 2 gives the basic type rules for $\Lambda B\sigma$. Aside from the introduction of explicit substitutions, these rules do not differ much from the original type rules in Figure 1. The type rule which is noticeable by its absence is a rule for typing closures. In fact since our substitutions are essentially untyped at the object level such a rule is not sound with respect to the original system. Instead (as with the second order $\lambda\sigma$-calculus [1]) we present rules for pushing substitutions inside of terms and typing the result; in general deciding well-typedness is inextricably tied with applying substitutions. We conjecture that such a closure rule would be sound in a system where dependent product types and LF-like encodings of terms were used to represent explicitly typed substitutions.

Figure 3 gives the rules for permuting substitutions with term constructors (including the ClosSubst rule for composing substitutions). Figure 5 gives the equivalence rules for substitutions, including rules for pushing substitutions inside of other substitutions. Here again we have a rule (MapSubst) for composing substitutions, analogous to ClosSubst.

The rules Beta, ClosConst, ClosVar ClosL, ClosR, ClosPi, ClosAbs, ClosApp, ClosSubst, MapTerm, MapComp and MapSubst constitute a higher-order rewrite system (HRS) as defined by Nipkow[5][18]. We now follow a line of reasoning similar to that for the $\lambda\sigma$-calculus [1] to verify the confluence of this system. To this purpose we separate the HRS into two subsystems: ΛB (constituting of only the Beta rule) and $\Lambda\sigma$ (constituting of the remaining rules). We denote (untyped) reduction under $\Lambda B\sigma$, ΛB and $\Lambda\sigma$ by $\rightarrow_{B\sigma}$, $\rightarrow_B$ and $\rightarrow_\sigma$, respectively. Recall that untyped β-reduction over terms of $\Lambda\beta$ is denoted by $\rightarrow_\beta$.

The type rules for object-language terms are given relative to a type environment Γ, with any free variables in an object language term bound in Γ. When considering the term equivalence rules as a HRS, the meta-variables in the schematic rewrite rules are considered free and the "free" object language variables are λ-bound in the metalanguage representation. When reasoning about the correctness of the HRS, we will assume that there are no free meta-variables in terms (any free object language variables are bound in Γ). A metalanguage term M with free variables in Γ may be considered as an abbreviation for $\lambda\Gamma \cdot M$, so in this sense we are restricting ourselves to "closed" terms.

Lemma 4.1 (Termination of $\Lambda\sigma$) *The HRS $\Lambda\sigma$ is Noetherian i.e. terminating.*

Proof: We adapt Field's termination proof for Accl. To reason about termination we will use a lexicographic semantic path ordering, although with a slightly non-standard approach. In particular we assume given, in addition to the usual term constructors, a countably infinite set of variables $\mathcal{X} = \{x_i\}_{i\in\omega}$ from which all λ-bound variables are taken. We define the following precedence on constructors:

$$\texttt{clos} \approx_o \texttt{map} \succ_o \texttt{apply} \succ_o \texttt{abs} \succ_o \texttt{pi} \succ_o _ \circ _ \succ_o [_,_]$$

The atomic terms are of the form $\pi_{i_1}(\ldots(\pi_{i_n}(x))\ldots)$ for $x \in \mathcal{X}$; we make these equivalent under the equivalence $\approx_o$and less than all of the other constructurs under the precedence $\preceq_o$.

[5]With the generalization that patterns are extended to products, and the restriction that right-hand sides are also patterns. We conjecture that Nipkow's Higher Order Critical Pairs Lemma still holds for this system.

The following measure gives a rough estimate of the eventual size of a term or substitution after normalization of substitutions:

$$
\begin{aligned}
|\, x \,| &\stackrel{\text{def}}{=} 1 \\
|\, \pi_i(M) \,| &\stackrel{\text{def}}{=} |\, M \,| \\
|\, \mathtt{apply}(M; N) \,| &\stackrel{\text{def}}{=} |\, M \,| + |\, N \,| + 1 \\
|\, \mathtt{abs}(A; \lambda x \cdot M) \,| &\stackrel{\text{def}}{=} |\, A \,| + |\, M \,| + 1 \\
|\, \mathtt{pi}(A; \lambda x \cdot B) \,| &\stackrel{\text{def}}{=} |\, A \,| + |\, B \,| + 1 \\
|\, \mathtt{clos}(\lambda x \cdot M; s) \,| &\stackrel{\text{def}}{=} |\, M \,| \cdot |\, s \,| \\
|\, [M, A] \,| &\stackrel{\text{def}}{=} |\, M \,| \\
|\, s_1 \circ s_2 \,| &\stackrel{\text{def}}{=} \mathtt{max}(|\, s_1 \,|.|\, s_2 \,|) \\
|\, \mathtt{map}(\lambda x \cdot s_1; s_2) \,| &\stackrel{\text{def}}{=} |\, s_1 \,| \cdot |\, s_2 \,|
\end{aligned}
$$

For object language terms $M \equiv \mathsf{t}_1(\overline{M_m})$ and $N \equiv \mathsf{t}_2(\overline{N_n})$, define the precedence ordering $M \succeq_t N$ by the lexicographic combination of $\succeq_o$ and eventual size under σ-normalization:

$$M \succ_t N \iff \mathsf{t}_1 \succ_o \mathsf{t}_2 \vee (\mathsf{t}_1 \approx_o \mathsf{t}_2 \wedge |\, M \,| > |\, N \,|)$$

Finally $\succeq_t$ is extended to a simplification ordering $\succeq$:

$$M \equiv \mathsf{t}_1(\overline{M_m}) \;\succ\; N \equiv \mathsf{t}_2(\overline{N_n})$$

if

1. $M_i \succeq N$ for some $i \in [m]$, or
2. $M \succ_t N$ and $M \succ N_j$ for all $j \in [n]$, or
3. $M \approx_t N$, $(M_1, \ldots, M_m) \succeq_* (N_1, \ldots, N_n)$ and $M \succ N_j$ for all $j \in [n]$.

Here $\succeq_*$ is the lexicographic extension of $\succeq$ to sequences, with $\lambda x \cdot M \succeq N$ if $M \succeq N$, $M \succeq \lambda x \cdot N$ if $M \succeq N$, and $\lambda x \cdot M \succeq \lambda x \cdot N$ if $M \succeq N$.

We can then verify that $\succ$ is a simplification ordering, and that $M \succ N$ where M and N are left and right hand sides, respectively, of any rule in the HRS.

□

Lemma 4.2 (Confluence of $\Lambda\sigma$) *$\Lambda\sigma$ is confluent on σ-closed terms i.e. if $\Gamma \vdash_\sigma M \in A$, $M \rightarrow_\sigma^* N_1$ and $M \rightarrow_\sigma^* N_2$, then there exists an N such that $N_1 \rightarrow_\sigma^* N$ and $N_2 \rightarrow_\sigma^* N$.*

PROOF: We verify local confluence by an examination of higher order critical pairs [18]. Confluence then follows from termination. The difficult case is for the critical pair formed by CLOSL and CLOSSUBST in

$\texttt{clos}(\lambda x \cdot \texttt{clos}(M(\pi_1(x)); s_1(\pi_1(x))); s_2 \circ s_2')$

We verify by induction on (maximal length of σ-reduction sequences, size of term) that:

1. $\texttt{map}(\lambda x \cdot s(\pi_1(x)); s_1 \circ s_2) = \texttt{map}(s; s_1)$
2. $\texttt{clos}(\lambda x \cdot M(\pi_1(x); \pi_2(x)); s_2 \circ \texttt{map}(s_1; s_2)) =$
   $\texttt{clos}(\lambda y \cdot M(\pi_1\pi_1(y); \pi_2(y)); (s_2 \circ s_2') \circ \texttt{map}(s_1; s_2))$

The base cases are for atomic terms of the form $M \equiv \lambda x \cdot \lambda y \cdot \pi_{i_1}(\ldots \pi_{i_n}(x)\ldots)$ for $z \in \{x, y\}$, $M \equiv \lambda x \cdot \lambda y \cdot z$ for $z \notin \{x, y\}$, and $M \equiv \lambda x \cdot \lambda y \cdot \texttt{type}(i)$. □

We now let $\sigma(M)$ ($\sigma(s)$) denote the (unique) normal form for the term M (substitution s) under the $\Lambda\sigma$ HRS. The remainder of the proof of confluence for $\Lambda B\sigma$ follows very closely that for the $\lambda\sigma$-calculus, in particular using Hardin's interpretation technique and confluence for $\Lambda\beta$.

We verify that the HRS $\Lambda B\sigma$ is a correct implementation of substitution. The following rules are for the judgement form $\Gamma \rhd \{N/x\}M \Longrightarrow M'$:

$$\textsc{Var}_1 \quad \frac{}{\Gamma, y : A, \Gamma' \rhd \{N/x\}y \Longrightarrow y} \quad x \neq y$$

$$\textsc{Var}_2 \quad \frac{}{\Gamma \rhd \{N/x\}x \Longrightarrow N}$$

$$\textsc{Type} \quad \frac{}{\Gamma \rhd \{N/x\}\texttt{Type}(i) \Longrightarrow \texttt{Type}(i)}$$

$$\textsc{App} \quad \frac{\Gamma \rhd \{N/x\}M_1 \Longrightarrow M_1' \quad \Gamma \rhd \{N/x\}M_2 \Longrightarrow M_2'}{\Gamma \rhd \{N/x\}\texttt{apply}(M_1, M_2) \Longrightarrow \texttt{apply}(M_1', M_2')}$$

$$\textsc{Abs} \quad \frac{\Gamma \rhd \{N/x\}A \Longrightarrow A' \quad \Gamma, y : A' \rhd \{N/x\}(M(y)) \Longrightarrow M'(y)}{\Gamma \rhd \{N/x\}\texttt{abs}(A; M) \Longrightarrow \texttt{abs}(A'; M')}$$

$$\textsc{Pi} \quad \frac{\Gamma \rhd \{N/x\}A \Longrightarrow A' \quad \Gamma, y : A' \rhd \{N/x\}(B(y)) \Longrightarrow B'(y)}{\Gamma \rhd \{N/x\}\texttt{pi}(A; B) \Longrightarrow \texttt{pi}(A'; B')}$$

We can then verify the following lemma by induction on the structure of a term M (or equivalently by induction on a derivation in the inference system just defined):

Lemma 4.3 *Suppose* $\Gamma, x : A \vdash_{ECC} M \in B$. *If* $\Gamma \vdash \{N/x\}M \Longrightarrow M'$ *then*

$$\sigma(\texttt{clos}(\lambda x \cdot M; [N, A])) \;=\; M'$$

Corollary 4.1 *Suppose* $\Gamma \vdash_{ECC} M \in A$ *and* $\Gamma \vdash_{ECC} N \in A$. *If* $M \to_\beta N$ *then* $M \to^*_{B\sigma} N$.

PROOF: By the definition of β-reduction and the previous lemma, it suffices to perform a Beta-reduction and then normalize with respect to $\Lambda\sigma$i.e. if $M \to_\beta N$ then $\exists M' \cdot M \to_B M'$ and $M' \to^*_\sigma N$. □

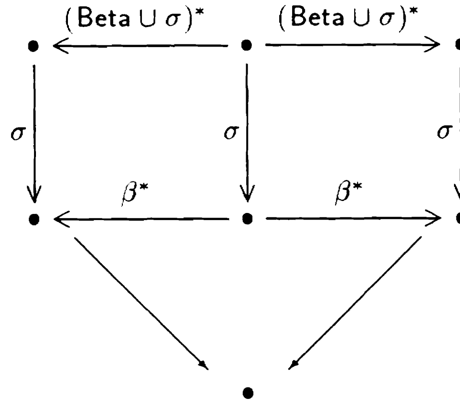
Corollary 4.2 *β-reduction is confluent on $\Lambda\sigma$ normal forms.*

Lemma 4.4

1. *For closed terms M and N, if $M \rightarrow_B N$ then $\sigma(M) \rightarrow_\beta^* \sigma(N)$.*
2. *For closed substitutions s and t, if $s \rightarrow_B t$ then $\sigma(s) \rightarrow_\beta^* \sigma(t)$.*

Theorem 1 *$\Lambda B\sigma$ is confluent on closed terms.*

PROOF: Using Hardin's interpretation technique [1] and Lemma 4.1, Lemma 4.2, Corollary 4.2 and Lemma 4.4. Hardin's technique amounts to verifying the following diagram (where the vertical arrows represent reduction to σ-normal form):



□

This crucial result is the basis for the type-checking algorithm presented in the next section. Finally we formulate a statement of correctness for $\Lambda B\sigma$ relative to $\Lambda\beta$:

Theorem 2 (Soundness of $\Lambda B\sigma$)

1. *If $\vdash_\sigma \Gamma \sim \Gamma'$ **env** then $\sigma(\Gamma) \equiv \sigma(\Gamma')$ and $\vdash_{ECC} \sigma(\Gamma)$ **env**.*
2. *If $\Gamma \vdash_\sigma M \sim N \in A$ then $\sigma(\Gamma) \vdash_{ECC} \sigma(M) = \sigma(N) \in \sigma(A)$.*
3. *If $\Gamma \vdash_\sigma M \preceq N \in A$ then $\sigma(\Gamma) \vdash_{ECC} \sigma(M) \preceq \sigma(N) \in \sigma(A)$.*
4. *If $\Gamma \vdash_\sigma s \sim t \in \texttt{Subst}(S)$ then*
   - *(a) if $S \equiv \texttt{Term}$ then $\sigma(s) \equiv [M, A] \equiv \sigma(t)$ for some M, A such that $\sigma(\Gamma) \vdash_{ECC} M \in A$.*
   - *(b) otherwise $S \equiv (S_1 \times S_2)$ for some S_1, S_2; then $\sigma(s) \equiv s_1 \circ s_2 \equiv \sigma(t)$ for some s_1, s_2 such that $\Gamma \vdash_\sigma s_1 \in \texttt{Subst}(S_1)$ and $\Gamma \vdash_\sigma s_2 \in \texttt{Subst}(S_2)$.*

It is unclear how to obtain an analogous completeness result. Although it has been suggested that this can be done for the $\lambda\sigma$-calculus by rewriting closures to BETA-redices, this does not seem to adequately handle definitional equality in the type system.

5 A Type-Checking Algorithm

Finally we briefly present a type-checking algorithm for $\Lambda\beta$ based on the system presented in the previous section, and state without proof the conditions for its correctness. Figure 6 presents the type-checker as a collection of inference rules, where closures are type-checked essentially by pushing substitutions inside of terms and type-checking the result.

These rules use numerous auxiliary algorithms. The rules for checking for cumulativity and convertibility (in Figures 7 and 8, respectively) are very similar, and amount to interleaving reductions to WHNF with recursive checking of subterms. Figures 9 and 11 give the algorithms for reducing terms and substitutions, respectively, to WHNF. Finally Figure 10 gives the algorithm for type-checking substitutions.

In contrast to the type system of $\Lambda B\sigma$, the type-checking algorithm does not "validate" the environment for each use of a variable. Rather it assumes an initial valid environment and then maintains the validity of the environment as terms are added to it. Also the WHNF reduction algorithms do not type-check their result, and rely for their correctness on the following:

Lemma 5.1

1. *If* $\Gamma \vdash_\sigma M \in A$ *and* $M \rightarrow^*_{B\sigma} N$ *then* $\Gamma \vdash_\sigma N \in A$.
2. *If* $\Gamma \vdash_\sigma s \in \mathtt{Subst}(S)$ *and* $s \rightarrow^*_{B\sigma} t$ *then* $\Gamma \vdash_\sigma t \in \mathtt{Subst}(S)$.
3. *If* $\Gamma \vdash_\sigma M \in A$ *then* $\Gamma \vdash_\sigma A \in \mathtt{Type}(i)$.
4. *If* $\Gamma \vdash_\sigma \mathtt{clos}(M; s) \in A$ *then* $\Gamma \vdash_\sigma s \in \mathtt{Subst}(S)$ *for some* S.

For the judgement forms $\Gamma \rhd M \in A$, $\Gamma \rhd A \preceq A'$, $\Gamma \rhd M \leftrightarrow N$, $\Gamma \rhd M \rightsquigarrow N$, $\Gamma \rhd s \in \mathtt{Subst}(S)$ and $\Gamma \rhd s \rightsquigarrow t$, let $\Gamma \vdash_{alg} M \in A$, $\Gamma \vdash_{alg} A \preceq A'$, $\Gamma \vdash_{alg} M \leftrightarrow N$, $\Gamma \vdash_{alg} M \rightsquigarrow N$, $\Gamma \vdash_{alg} s \in \mathtt{Subst}(S)$ and $\Gamma \vdash_{alg} s \rightsquigarrow t$, respectively, denote derivability according to the inference rules of the type-checking algorithm. The statement of soundness for the type-checker is then given by

Theorem 3 *Suppose* $\vdash_\sigma \Gamma \sim \Gamma$ **env**. *Then:*

1. *If* $\Gamma \vdash_{alg} M \in A$ *then* $\Gamma \vdash_\sigma M \in A$.
2. *If* $\Gamma \vdash_{alg} A \in \mathtt{Type}(i)$, $\Gamma \vdash_{alg} A' \in \mathtt{Type}(i)$ *and* $\Gamma \vdash_{alg} A \preceq A'$ *then* $\Gamma \vdash_\sigma A \preceq A' \in \mathtt{Type}(i)$.
3. *If* $\Gamma \vdash_{alg} M \in A$, $\Gamma \vdash_{alg} N \in A$ *and* $\Gamma \vdash_{alg} M \leftrightarrow N$ *then* $\Gamma \vdash_\sigma M \sim N \in A$.
4. *If* $\Gamma \vdash_{alg} M \in A$ *and* $\Gamma \vdash_{alg} M \rightsquigarrow N$ *then* $\Gamma \vdash_\sigma M \sim N \in A$.
5. *If* $\Gamma \vdash_{alg} s \in \mathtt{Subst}(S)$ *then* $\Gamma \vdash_\sigma s \in \mathtt{Subst}(S)$.
6. *If* $\Gamma \vdash_{alg} s \in \mathtt{Subst}(S)$ *and* $\Gamma \vdash_{alg} s \rightsquigarrow t$ *then* $\Gamma \vdash_\sigma s \sim t \in \mathtt{Subst}(S)$.

6 Conclusions

We have presented an approach to incorporating explicit substitutions into L_λ-like languages, based on a generalization of Miller's patterns to product types. Although there appears to be some promise with the approach, ultimately its usefulness may depend on implementational considerations. In particular the form of restricted β-reductions allowed in the metalanguage appear somewhat more complicated to implement than β_0-reduction [15]. Although there are advantages to having substitutions outside of the inference engine in λ-Prolog-like languages, it remains to be seen what the performance penalty for this might be. However provided this performance penalty is not too great, there are important pragmatic advantages to providing substitutions outside of the programming language evaluator. Among these are that applications that do not use substitutions should not pay the price for their provision, and also that applications may be provided in a more flexible way (e.g. as "classes") allowing them to be tailored for specific applications. This is important for example in providing "defined constants" in a theorem-proving environment, which are crucial for controlling the size of terms during comparison and printing.

Acknowledgement: Paul Taylor's diagram package was used to draw the diagram on Page 9.

7 Appendix

EnvNil	$\dfrac{}{\textsf{nil}\ \mathbf{env}}$	
EnvExt	$\dfrac{\Gamma \rhd A = A' \in \textsf{Type}(i)}{\Gamma, x : A\ \mathbf{env}}$	$(x \text{ new})$
Var	$\dfrac{\Gamma, x : A, \Gamma'\ \mathbf{env}}{\Gamma, x : A, \Gamma' \rhd x = x \in A}$	
Type	$\dfrac{\Gamma\ \mathbf{env}}{\Gamma \rhd \textsf{Type}(i) = \textsf{Type}(i) \in \textsf{Type}(s(i))}$	
Pi	$\dfrac{\Gamma \rhd A = A' \in \textsf{Type}(i) \quad \Gamma, x : A \rhd B(x) = B'(x) \in \textsf{Type}(i)}{\Gamma \rhd \textsf{pi}(A; B) = \textsf{pi}(A'; B') \in \textsf{Type}(i)}$	
Abs	$\dfrac{\Gamma \rhd A = A' \in \textsf{Type}(i) \quad \Gamma, x : A \rhd M(x) = M'(x) \in B(x)}{\Gamma \rhd \textsf{abs}(A; M) = \textsf{abs}(A'; M') \in \textsf{pi}(A; B)}$	
App	$\dfrac{\Gamma \rhd M = M' \in \textsf{pi}(A; B) \quad \Gamma \rhd N = N' \in A}{\Gamma \rhd \textsf{apply}(M; N) = \textsf{apply}(M'; N') \in B(N)}$	
Beta	$\dfrac{\Gamma, x : A \rhd M(x) \in B(x) \qquad \Gamma \rhd N \in A}{\Gamma \rhd \textsf{apply}(\textsf{abs}(A; M); N) = M(N) \in B(N)}$	
Cum	$\dfrac{\Gamma \rhd M = M' \in A \quad \Gamma \rhd A' \in \textsf{Type}(i) \quad \Gamma \rhd A \preceq A' \in \textsf{Type}(i)}{\Gamma \rhd M = M' \in A'}$	
Sym	$\dfrac{\Gamma \rhd M = M' \in A}{\Gamma \rhd M' = M \in A}$	
Trans	$\dfrac{\Gamma \rhd M_1 = M_2 \in A \quad \Gamma \rhd M_2 = M_3 \in A}{\Gamma \rhd M_1 = M_3 \in A}$	
CumEq	$\dfrac{\Gamma \rhd A = A' \in \textsf{Type}(i)}{\Gamma \rhd A \preceq A' \in \textsf{Type}(i)}$	
CumTrans	$\dfrac{\Gamma \rhd A_1 \preceq A_2 \in \textsf{Type}(i) \quad \Gamma \rhd A_2 \preceq A_3 \in \textsf{Type}(i)}{\Gamma \rhd A_1 \preceq A_3 \in \textsf{Type}(i)}$	
CumType	$\dfrac{\Gamma\ \mathbf{env}}{\Gamma \rhd \textsf{Type}(i) \preceq \textsf{Type}(s(i)) \in \textsf{Type}(j)}$	$(s(i) < j)$
CumPi	$\dfrac{\Gamma \rhd A' \preceq A \in \textsf{Type}(i) \quad \Gamma, x : A' \rhd B(x) \preceq B'(x) \in \textsf{Type}(i)}{\Gamma \rhd \textsf{pi}(A; B) \preceq \textsf{pi}(A'; B') \in \textsf{Type}(i)}$	

Figure 1: $\Lambda\beta$: Luo's Extended Calculus of Constructions

$$\textsc{Var}\quad \frac{\Gamma, x : A, \Gamma' \ \mathbf{env}}{\Gamma, x : A, \Gamma' \rhd x \sim x \in A}$$

$$\textsc{Type}\quad \frac{\Gamma \ \mathbf{env}}{\Gamma \rhd \mathtt{Type}(i) \sim \mathtt{Type}(i) \in \mathtt{Type}(s(i))}$$

$$\textsc{Pi}\quad \frac{\Gamma \rhd A \sim A' \in \mathtt{Type}(i) \quad \Gamma, x : A \rhd B(x) \sim B'(x) \in \mathtt{Type}(i)}{\Gamma \rhd \mathtt{pi}(A; B) \sim \mathtt{pi}(A'; B') \in \mathtt{Type}(i)}$$

$$\textsc{Abs}\quad \frac{\Gamma \rhd A \sim A' \in \mathtt{Type}(i) \quad \Gamma, x : A \rhd M(x) \sim M'(x) \in B(x)}{\Gamma \rhd \mathtt{abs}(A; M) \sim \mathtt{abs}(A'; M') \in \mathtt{pi}(A; B)}$$

$$\textsc{App}\quad \frac{\Gamma \rhd M \sim M' \in \mathtt{pi}(A; B) \qquad \Gamma \rhd N \sim N' \in A}{\Gamma \rhd \mathtt{apply}(M; N) \sim \mathtt{apply}(M'; N') \in \mathtt{clos}(B; [N, A])}$$

$$\textsc{Beta}\quad \frac{\Gamma, x : A \rhd M(x) \sim M(x) \in B(x) \qquad \Gamma \rhd N \sim N \in A}{\Gamma \rhd \mathtt{apply}(\mathtt{abs}(A; M); N) \sim \mathtt{clos}(M; [N, A]) \in \mathtt{clos}(B; [N, A])}$$

$$\textsc{Cum}\quad \frac{\Gamma \rhd M \sim M' \in A \quad \Gamma \rhd A' \sim A' \in \mathtt{Type}(i) \quad \Gamma \rhd A \preceq A' \in \mathtt{Type}(i)}{\Gamma \rhd M \sim M' \in A'}$$

$$\textsc{Env}\quad \frac{\Gamma \rhd M \sim M' \in A \quad \Gamma \sim \Gamma' \ \mathbf{env}}{\Gamma' \rhd M \sim M' \in A}$$

$$\textsc{Sym}\quad \frac{\Gamma \rhd M \sim M' \in A}{\Gamma \rhd M' \sim M \in A}$$

$$\textsc{Trans}\quad \frac{\Gamma \rhd M_1 \sim M_2 \in A \quad \Gamma \rhd M_2 \sim M_3 \in A}{\Gamma \rhd M_1 \sim M_3 \in A}$$

$$\textsc{CumEq}\quad \frac{\Gamma \rhd A \sim A' \in \mathtt{Type}(i)}{\Gamma \rhd A \preceq A' \in \mathtt{Type}(i)}$$

$$\textsc{CumTrans}\quad \frac{\Gamma \rhd A_1 \preceq A_2 \in \mathtt{Type}(i) \quad \Gamma \rhd A_2 \preceq A_3 \in \mathtt{Type}(i)}{\Gamma \rhd A_1 \preceq A_3 \in \mathtt{Type}(i)}$$

$$\textsc{CumType}\quad \frac{\Gamma \ \mathbf{env}}{\Gamma \rhd \mathtt{Type}(i) \preceq \mathtt{Type}(s(i)) \in \mathtt{Type}(j)} \quad (s(i) < j)$$

$$\textsc{CumPi}\quad \frac{\Gamma \rhd A' \preceq A \in \mathtt{Type}(i) \quad \Gamma, x : A' \rhd B(x) \preceq B'(x) \in \mathtt{Type}(i)}{\Gamma \rhd \mathtt{pi}(A; B) \preceq \mathtt{pi}(A'; B') \in \mathtt{Type}(i)}$$

Figure 2: $\Lambda B\sigma$: ECC With Explicit Substitutions

$$\textsc{ClosCong}\quad \frac{\Gamma \rhd s \sim s' \in \mathtt{Subst}(S) \quad \Gamma \rhd \mathtt{clos}(M; s') \in A}{\Gamma \rhd \mathtt{clos}(M; s) \sim \mathtt{clos}(M; s') \in A}$$

$$\textsc{ClosConst}\quad \frac{\Gamma \rhd s \in \mathtt{Subst}(S) \quad \Gamma \rhd M \in A}{\Gamma \rhd \mathtt{clos}(\lambda y \cdot M; s) \sim M \in A}$$

$$\textsc{ClosVar}\quad \frac{\Gamma \rhd [M, A] \in \mathtt{Subst(Term)}}{\Gamma \rhd \mathtt{clos}(\lambda y \cdot y; [M, A]) \sim M \in A}$$

$$\textsc{ClosL}\quad \frac{\Gamma \rhd s_2 \in \mathtt{Subst}(S) \qquad \Gamma \rhd \mathtt{clos}(M; s_1) \in B}{\Gamma \rhd \mathtt{clos}(\lambda x \cdot M(\pi_1(x)); s_1 \circ s_2) \sim \mathtt{clos}(M; s_1) \in B}$$

$$\textsc{ClosR}\quad \frac{\Gamma \rhd s_1 \in \mathtt{Subst}(S) \qquad \Gamma \rhd \mathtt{clos}(M; s_2) \in B}{\Gamma \rhd \mathtt{clos}(\lambda x \cdot M(\pi_2(x)); s_1 \circ s_2) \sim \mathtt{clos}(M; s_2) \in B}$$

$$\textsc{ClosPi}\quad \frac{\Gamma \rhd \mathtt{pi}(\mathtt{clos}(A; s); \lambda y \cdot \mathtt{clos}(\lambda x \cdot B(x; y); s)) \in C}{\Gamma \rhd \mathtt{clos}(\lambda x \cdot \mathtt{pi}(A(x); B(x)); s) \sim \mathtt{pi}(\mathtt{clos}(A; s); \lambda y \cdot \mathtt{clos}(\lambda x \cdot B(x; y); s)) \in C}$$

$$\textsc{ClosAbs}\quad \frac{\Gamma \rhd \mathtt{abs}(\mathtt{clos}(A; s); \lambda y \cdot \mathtt{clos}(\lambda x \cdot M(x; y); s)) \in C}{\Gamma \rhd \mathtt{clos}(\lambda x \cdot \mathtt{abs}(A(x); M(x)); s) \sim \mathtt{abs}(\mathtt{clos}(A; s); \lambda y \cdot \mathtt{clos}(\lambda x \cdot M(x; y); s)) \in C}$$

$$\textsc{ClosApp}\quad \frac{\Gamma \rhd \mathtt{apply}(\mathtt{clos}(M; s); \mathtt{clos}(N; s)) \in C}{\Gamma \rhd \mathtt{clos}(\lambda x \cdot \mathtt{apply}(M(x); N(x)); s) \sim \mathtt{apply}(\mathtt{clos}(M; s); \mathtt{clos}(N; s)) \in C}$$

$$\textsc{ClosSubst}\quad \frac{\Gamma \rhd \mathtt{clos}(\lambda x \cdot M(\pi_1(x); \pi_2(x)); s_2 \circ \mathtt{map}(s_1; s_2)) \in C}{\Gamma \rhd \mathtt{clos}(\lambda x \cdot \mathtt{clos}(M(x); s_1(x)); s_2) \sim \mathtt{clos}(\lambda x \cdot M(\pi_1(x); \pi_2(x)); s_2 \circ \mathtt{map}(s_1; s_2)) \in C}$$

Figure 3: $\Lambda B\sigma$(cont'd): Substitution Rules for Terms

$$\textsc{EnvNil}\quad \frac{}{\texttt{nil} \sim \texttt{nil}\ \textbf{env}}$$

$$\textsc{EnvExt}\quad \frac{\Gamma \sim \Gamma'\ \textbf{env} \quad \Gamma \rhd A \sim A' \in \mathsf{Type}(i)}{(\Gamma, x : A) \sim (\Gamma', x : A')\ \textbf{env}} \quad (x \text{ new})$$

$$\textsc{EnvSym}\quad \frac{\Gamma \sim \Gamma'\ \textbf{env}}{\Gamma' \sim \Gamma\ \textbf{env}}$$

$$\textsc{EnvTrans}\quad \frac{\Gamma_1 \sim \Gamma_2\ \textbf{env} \quad \Gamma_2 \sim \Gamma_3\ \textbf{env}}{\Gamma_1 \sim \Gamma_3\ \textbf{env}}$$

Figure 4: $\Lambda B\sigma$(cont'd): Environment Equivalence Rules

$$\textsc{MapCong}\quad \frac{\Gamma \rhd s_2 \sim s_2' \in \texttt{Subst}(S_2) \quad \Gamma \rhd \texttt{map}(s_1; s_2') \in \texttt{Subst}(S_1)}{\Gamma \rhd \texttt{map}(s_1; s_2) \sim \texttt{map}(s_1; s_2') \in \texttt{Subst}(S_1)}$$

$$\textsc{SubstTerm}\quad \frac{\Gamma \rhd A \sim B \in \mathsf{Type}(i) \quad \Gamma \rhd M \sim N \in A}{\Gamma \rhd [M, A] \sim [N, B] \in \texttt{Subst(Term)}}$$

$$\textsc{SubstComp}\quad \frac{\Gamma \rhd s_1 \sim t_1 \in \texttt{Subst}(S_1) \quad \Gamma \rhd s_2 \sim t_2 \in \texttt{Subst}(S_2)}{\Gamma \rhd s_1 \circ s_2 \sim t_1 \circ t_2 \in \texttt{Subst}(S_1 \times S_2)}$$

$$\textsc{MapTerm}\quad \frac{\Gamma \rhd [\texttt{clos}(M; s), \texttt{clos}(A; s)] \in \texttt{Subst(Term)}}{\Gamma \rhd \texttt{map}(\lambda x \cdot [M(x), A(x)]; s) \sim [\texttt{clos}(M; s), \texttt{clos}(A; s)] \in \texttt{Subst(Term)}}$$

$$\textsc{MapComp}\quad \frac{\Gamma \rhd \texttt{map}(s_1; s) \circ \texttt{map}(s_2; s) \in \texttt{Subst}(S_1 \times S_2)}{\Gamma \rhd \texttt{map}(\lambda x \cdot s_1(x) \circ s_2(x); s) \sim \texttt{map}(s_1; s) \circ \texttt{map}(s_2; s) \in \texttt{Subst}(S_1 \times S_2)}$$

$$\textsc{MapSubst}\quad \frac{\Gamma \rhd \texttt{map}(\lambda x \cdot s_1(\pi_1(x); \pi_2(x)); s \circ \texttt{map}(s_2; s)) \in \texttt{Subst}(S)}{\Gamma \rhd \texttt{map}(\lambda x \cdot \texttt{map}(s_1(x); s_2(x)); s) \sim \texttt{map}(\lambda x \cdot s_1(\pi_1(x); \pi_2(x)); s \circ \texttt{map}(s_2; s)) \in \texttt{Subst}(S)}$$

$$\textsc{SubstSym}\quad \frac{\Gamma \rhd s \sim t \in \texttt{Subst}(S)}{\Gamma \rhd t \sim s \in \texttt{Subst}(S)}$$

$$\textsc{SubstTrans}\quad \frac{\Gamma \rhd s_1 \sim s_2 \in \texttt{Subst}(S) \quad \Gamma \rhd s_2 \sim s_3 \in \texttt{Subst}(S)}{\Gamma \rhd s_1 \sim s_3 \in \texttt{Subst}(S)}$$

$$\textsc{SubstEnv}\quad \frac{\Gamma \rhd s \sim t \in \texttt{Subst}(S) \quad \Gamma \sim \Gamma'\ \textbf{env}}{\Gamma' \rhd s \sim t \in \texttt{Subst}(S)}$$

Figure 5: $\Lambda B\sigma$(cont'd): Equivalence Rules for Substitutions

VAR
$$\frac{}{\Gamma, x : A, \Gamma' \rhd x \in A}$$

TYPE
$$\frac{}{\Gamma \rhd \mathtt{Type}(i) \in \mathtt{Type}(s(i))}$$

PI
$$\frac{\begin{array}{cc}\Gamma \rhd A \in A' & \Gamma, x : A \rhd B(x) \in B'(x) \\ \Gamma \rhd A' \leadsto \mathtt{Type}(i) & \Gamma, x : A \rhd B'(x) \leadsto \mathtt{Type}(j)\end{array}}{\Gamma \rhd \mathtt{pi}(A, B) \in \mathtt{Type}(\mathtt{max}(i, j))}$$

ABS
$$\frac{\Gamma \rhd A \in A' \quad \Gamma \rhd A' \leadsto \mathtt{Type}(i) \quad \Gamma, x : A \rhd M(x) \in B(x)}{\Gamma \rhd \mathtt{abs}(A, M) \in \mathtt{pi}(A, B)}$$

APP
$$\frac{\Gamma \rhd M \in B' \quad \Gamma \rhd B' \leadsto \mathtt{pi}(A, B) \quad \Gamma \rhd N \in A' \quad \Gamma \rhd A' \preceq A}{\Gamma \rhd \mathtt{apply}(M; N) \in \mathtt{clos}(B; [N, A'])}$$

CLOS
$$\frac{\Gamma \rhd s \in \mathtt{Subst}(S) \quad \Gamma \rhd M \in A}{\Gamma \rhd \mathtt{clos}(\lambda x \cdot M; s) \in A}$$

CLOSVAR
$$\frac{\Gamma \rhd s \leadsto [M, A] \quad \Gamma \rhd M \in A}{\Gamma \rhd \mathtt{clos}(\lambda x \cdot x; [M, A]) \in A}$$

CLOSL
$$\frac{\Gamma \rhd s \leadsto s_1 \circ s_2 \quad \Gamma \rhd s_2 \in \mathtt{Subst}(S) \quad \Gamma \rhd \mathtt{clos}(M; s_1) \in A}{\Gamma \rhd \mathtt{clos}(\lambda x \cdot M(\pi_1(x)); s) \in A}$$

CLOSR
$$\frac{\Gamma \rhd s \leadsto s_1 \circ s_2 \quad \Gamma \rhd s_1 \in \mathtt{Subst}(S) \quad \Gamma \rhd \mathtt{clos}(M; s_2) \in A}{\Gamma \rhd \mathtt{clos}(\lambda x \cdot M(\pi_2(x)); s) \in A}$$

CLOSPI
$$\frac{\begin{array}{cc}\Gamma \rhd \mathtt{clos}(A; s) \in A' & \Gamma, x : A \rhd \mathtt{clos}(B(x); s) \in B'(x) \\ \Gamma \rhd A' \leadsto \mathtt{Type}(i) & \Gamma, x : A \rhd B'(x) \leadsto \mathtt{Type}(j)\end{array}}{\Gamma \rhd \mathtt{clos}(\lambda y \cdot \mathtt{pi}(A(y), B(y)); s) \in \mathtt{Type}(\mathtt{max}(i, j)}$$

CLOSABS
$$\frac{\Gamma \rhd \mathtt{clos}(A; s) \in A' \quad \Gamma \rhd A' \leadsto \mathtt{Type}(i) \quad \Gamma, x : A \rhd \mathtt{clos}(M(x); s) \in B(x)}{\Gamma \rhd \mathtt{clos}(\lambda y \cdot \mathtt{abs}(A(y); M(y)); s) \in \mathtt{pi}(A; B)}$$

CLOSAPP
$$\frac{\Gamma \rhd \mathtt{clos}(M; s) \in B' \quad \Gamma \rhd B' \leadsto \mathtt{pi}(A; B) \quad \Gamma \rhd \mathtt{clos}(N; s) \in A' \quad \Gamma \rhd A' \preceq A}{\Gamma \rhd \mathtt{clos}(\lambda x \cdot \mathtt{apply}(M(x); N(x)); s) \in \mathtt{clos}(B; [\mathtt{clos}(N; s), A'])}$$

CLOSSUBST
$$\frac{\Gamma \rhd \mathtt{clos}(\lambda x \cdot M(\pi_1(x); \pi_2(x)); s_2 \circ \mathtt{map}(s_1; s_2)) \in C}{\Gamma \rhd \mathtt{clos}(\lambda x \cdot \mathtt{clos}(M(x); s_1(x)); s_2) \in C}$$

Figure 6: Type Checking Algorithm for ECC

CumType

$$\frac{\Gamma \rhd A \leadsto \mathtt{Type}(i) \quad \Gamma \rhd A' \leadsto \mathtt{Type}(j) \quad i \le j}{\Gamma \rhd A \preceq A'}$$

CumPi

$$\frac{\Gamma \rhd M \leadsto \mathtt{pi}(A;B) \quad \Gamma \rhd M' \leadsto \mathtt{pi}(A';B') \quad \Gamma \rhd A' \preceq A \quad \Gamma, x : A' \rhd B(x) \preceq B'(x)}{\Gamma \rhd \mathtt{pi}(A;B) \preceq \mathtt{pi}(A';B')}$$

CumAbs

$$\frac{\Gamma \rhd M_1 \leadsto \mathtt{abs}(A_1;M_1') \quad \Gamma \rhd M_2 \leadsto \mathtt{abs}(A_2;M_2') \quad \Gamma \rhd A_1 \leftrightarrow A_2 \quad \Gamma, x : A_1 \rhd M_1'(x) \leftrightarrow M_2'(x)}{\Gamma \rhd M_1 \preceq M_2}$$

CumApp

$$\frac{\Gamma \rhd M \leadsto \mathtt{apply}(M_1;M_2) \quad \Gamma \rhd N \leadsto \mathtt{apply}(N_1;N_2) \quad \Gamma \rhd M_1 \leftrightarrow N_1 \quad \Gamma \rhd M_2 \leftrightarrow N_2}{\Gamma \rhd M \preceq N}$$

CumVar

$$\frac{\Gamma, x : A, \Gamma' \rhd M \leadsto x \quad \Gamma, x : A, \Gamma' \rhd N \leadsto x}{\Gamma, x : A, \Gamma' \rhd M \preceq N}$$

Figure 7: Cumulativity Algorithm for ECC

EqType
$$\frac{\Gamma \rhd A \leadsto \mathtt{Type}(i) \quad \Gamma \rhd A' \leadsto \mathtt{Type}(i)}{\Gamma \rhd A \leftrightarrow A'}$$

EqPi
$$\frac{\Gamma \rhd M \leadsto \mathtt{pi}(A;B) \quad \Gamma \rhd M' \leadsto \mathtt{pi}(A';B') \quad \Gamma \rhd A' \leftrightarrow A \quad \Gamma, x : A' \rhd B(x) \leftrightarrow B'(x)}{\Gamma \rhd \mathtt{pi}(A;B) \leftrightarrow \mathtt{pi}(A';B')}$$

EqAbs
$$\frac{\Gamma \rhd M_1 \leadsto \mathtt{abs}(A_1;M_1') \quad \Gamma \rhd M_2 \leadsto \mathtt{abs}(A_2;M_2') \quad \Gamma \rhd A_1 \leftrightarrow A_2 \quad \Gamma, x : A_1 \rhd M_1'(x) \leftrightarrow M_2'(x)}{\Gamma \rhd M_1 \leftrightarrow M_2}$$

EqApp
$$\frac{\Gamma \rhd M \leadsto \mathtt{apply}(M_1;M_2) \quad \Gamma \rhd N \leadsto \mathtt{apply}(N_1;N_2) \quad \Gamma \rhd M_1 \leftrightarrow N_1 \quad \Gamma \rhd M_2 \leftrightarrow N_2}{\Gamma \rhd M \leftrightarrow N}$$

EqVar
$$\frac{\Gamma, x : A, \Gamma' \rhd M \leadsto x \quad \Gamma, x : A, \Gamma' \rhd N \leadsto x}{\Gamma, x : A, \Gamma' \rhd M \leftrightarrow N}$$

Figure 8: Term Equivalence Algorithm for ECC

RedVar
$$\frac{}{\Gamma, x : A, \Gamma' \rhd x \leadsto x}$$

RedType
$$\frac{}{\Gamma \rhd \mathtt{Type}(i) \leadsto \mathtt{Type}(i)}$$

RedPi
$$\frac{}{\Gamma \rhd \mathtt{pi}(A; B) \leadsto \mathtt{pi}(A; B)}$$

RedAbs
$$\frac{}{\Gamma \rhd \mathtt{abs}(A; M) \leadsto \mathtt{abs}(A; M)}$$

RedAppVar
$$\frac{\Gamma, x : A, \Gamma' \rhd M \leadsto x}{\Gamma, x : A, \Gamma' \rhd \mathtt{apply}(M; N) \leadsto \mathtt{apply}(x; N)}$$

RedAppAbs
$$\frac{\Gamma \rhd M \leadsto \mathtt{abs}(A; M') \quad \Gamma \rhd \mathtt{clos}(M'; [N, A]) \leadsto M''}{\Gamma \rhd \mathtt{apply}(M; N) \leadsto M''}$$

RedAppApp
$$\frac{\Gamma \rhd M \leadsto \mathtt{apply}(M'; N')}{\Gamma \rhd \mathtt{apply}(M; N) \leadsto \mathtt{apply}(\mathtt{apply}(M'; N'); N)}$$

RedClosConst
$$\frac{\Gamma \rhd M \leadsto M'}{\Gamma \rhd \mathtt{clos}(\lambda x \cdot M; s) \leadsto M'}$$

RedClosVar
$$\frac{\Gamma \rhd s \leadsto [M, A]}{\Gamma \rhd \mathtt{clos}(\lambda y \cdot y; s) \leadsto M}$$

RedClosL
$$\frac{\Gamma \rhd s \leadsto s_1 \circ s_2 \quad \Gamma \rhd \mathtt{clos}(\lambda x \cdot M(x); s_1) \leadsto M'}{\Gamma \rhd \mathtt{clos}(\lambda x \cdot M(\pi_1(x)); s) \leadsto M'}$$

RedClosR
$$\frac{\Gamma \rhd s \leadsto s_1 \circ s_2 \quad \Gamma \rhd \mathtt{clos}(\lambda x \cdot M(x); s_2) \leadsto M'}{\Gamma \rhd \mathtt{clos}(\lambda x \cdot M(\pi_2(x)); s) \leadsto M'}$$

RedClosPi
$$\frac{}{\Gamma \rhd \mathtt{clos}(\lambda x \cdot \mathtt{pi}(A(x); B(x)); s) \leadsto \mathtt{pi}(\mathtt{clos}(A; s); \lambda y \cdot \mathtt{clos}(\lambda x \cdot B(x; y); s))}$$

RedClosAbs
$$\frac{}{\Gamma \rhd \mathtt{clos}(\lambda x \cdot \mathtt{abs}(A(x); M(x)); s) \leadsto \mathtt{abs}(\mathtt{clos}(A; s); \lambda y \cdot \mathtt{clos}(\lambda x \cdot M(x; y); s))}$$

RedClosApp
$$\frac{}{\Gamma \rhd \mathtt{clos}(\lambda x \cdot \mathtt{apply}(M(x); N(x)); s) \leadsto \mathtt{apply}(\mathtt{clos}(M; s); \mathtt{clos}(N; s))}$$

RedClosComp
$$\frac{\Gamma \rhd \mathtt{clos}(\lambda x \cdot M(\pi_1(x); \pi_2(x)); s_2 \circ \mathtt{map}(s_1; s_2)) \leadsto M'}{\Gamma \rhd \mathtt{clos}(\lambda x \cdot \mathtt{clos}(M(x); s_1(x)); s_2) \leadsto M'}$$

Figure 9: WHNF Reduction Algorithm for Terms

SUBSTTERM
$$\frac{\Gamma \rhd A \in A' \quad \Gamma \rhd A' \leadsto \mathtt{Type}(i) \quad \Gamma \rhd M \in B \quad \Gamma \rhd A \leftrightarrow B}{\Gamma \rhd [M, A] \in \mathtt{Subst(Term)}}$$

SUBSTCOMP
$$\frac{\Gamma \rhd s_1 \in \mathtt{Subst}(S_1) \quad \Gamma \rhd s_2 \in \mathtt{Subst}(S_2)}{\Gamma \rhd s_1 \circ s_2 \in \mathtt{Subst}(S_1 \times S_2)}$$

MAPTERM
$$\frac{\Gamma \rhd \mathtt{clos}(A; s) \in A' \quad \Gamma \rhd A' \leadsto \mathtt{Type}(i) \quad \Gamma \rhd \mathtt{clos}(M; s) \in B \quad \Gamma \rhd \mathtt{clos}(A; s) \leftrightarrow B}{\Gamma \rhd \lambda x \cdot \mathtt{map}([M(x), A(x)]; s) \in \mathtt{Subst(Term)}}$$

MAPCOMP
$$\frac{\Gamma \rhd \mathtt{map}(s_1; s) \in \mathtt{Subst}(S_1) \quad \Gamma \rhd \mathtt{map}(s_2; s) \in \mathtt{Subst}(S_2)}{\Gamma \rhd \mathtt{map}(\lambda x \cdot s_1(x) \circ s_2(x); s) \in \mathtt{Subst}(S_1 \times S_2)}$$

MAPSUBST
$$\frac{\Gamma \rhd \mathtt{map}(\lambda x \cdot s_1(\pi_1(x); \pi_2(x)); s \circ \mathtt{map}(s_2; s)) \in \mathtt{Subst}(S)}{\Gamma \rhd \mathtt{map}(\lambda x \cdot \mathtt{map}(s_1(x); s_2(x)); s) \in \mathtt{Subst}(S)}$$

Figure 10: Type Inference for Substitutions

REDMAPTERM
$$\frac{}{\Gamma \rhd \mathtt{map}(\lambda x \cdot [M(x), A(x)]; s) \leadsto [\mathtt{clos}(M; s), \mathtt{clos}(A; s)]}$$

REDMAPCOMP
$$\frac{}{\Gamma \rhd \mathtt{map}(\lambda x \cdot s_1(x) \circ s_2(x); s) \leadsto \mathtt{map}(s_1; s) \circ \mathtt{map}(s_2; s)}$$

REDMAPCOMPOSE
$$\frac{\Gamma \rhd \mathtt{map}(\lambda x \cdot s_1(\pi_1(x); \pi_2(x)); s \circ \mathtt{map}(s_2; s)) \leadsto s'}{\Gamma \rhd \mathtt{map}(\lambda x \cdot \mathtt{map}(s_1(x); s_2(x)); s) \leadsto s'}$$

Figure 11: WHNF Reduction Algorithm for Substitutions

References

[1] Martin Abadi, Luca Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

[2] R. Burstall and B. W. Lampson. A kernel language for abstract data types and modules. In G. Kahn, D. B. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, pages 1–50. Springer-Verlag, 1984. Lecture Notes in Computer Science 173.

[3] Luca Cardelli. Typeful programming. Technical report, DEC Systems Research Center, 1989.

[4] P. Crégut. An abstract machine for the normalization of λ-terms. In *Proceedings of ACM Symposium on Lisp and Functional Programming*, pages 333–340, 1990.

[5] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Proceedings of the Koninklijke Nederlandse Akademie van Wetenschappen*, 75(5):381–392, 1972.

[6] D. Duggan. A type-theoretic framework for metaprogramming. In preparation, 1992.

[7] D. Duggan. Unification with extended patterns in ecc. In preparation., 1992.

[8] C. Elliott and F. Pfenning. Higher order abstract syntax. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1988.

[9] John Field. On laziness and optimality in lambda interpreters: Tools for specification and analysis. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 1–15, 1990.

[10] Georges Gonthier, Martin Abadi, and Jean-Jacques Lévy. The geometry of optimal λ-reduction. In *Proceedings of ACM Symposium on Principles of Programming Languages*, 1992.

[11] John Hannan and Dale Miller. From operational semantics to abstract machines (preliminary results). In *Proceedings of ACM Symposium on Lisp and Functional Programming*, 1990.

[12] R. Harper and R. Pollack. Type checking with universes. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development*, 1989.

[13] Joshua S. Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. In *Proceedings of IEEE Symposium on Logic in Computer Science*, 1991.

[14] Z. Luo. Ecc, an extended calculus of constructions. In *Proceedings of IEEE Symposium on Logic in Computer Science*, pages 385–395, 1989.

[15] D. A. Miller. A logic programming language with lambda-abstraction, function variables and simple unification. In P. Schroeder-Heister, editor, *Extensions of Logic Programming.* Springer-Verlag Lecture Notes in Computer Science, 1990.

[16] D. A. Miller and G. Nadathur. A logic programming approach to manipulating formulas and programs. In *Proceedings of the IEEE Symposium on Logic Programming*, 1987.

[17] Gopalan Nadathur and Debra Sue Wilson. A representation of lambda terms suitable for operations on their intensions. In *Proceedings of ACM Symposium on Lisp and Functional Programming*, pages 341–348, 1990.

[18] Tobias Nipkow. Higher order critical pairs. In *Proceedings of IEEE Symposium on Logic in Computer Science*, 1991.

[19] F. Pfenning. Unification and anti-unification in the calculus of constructions. In *Proceedings of IEEE Symposium on Logic in Computer Science*, 1991.

[20] F. Pfenning and P. Lee. Leap: A language with eval and polymorphism. In *TAPSOFT'89: Proceedings of the International Joint Conference on Theory and Practice in Software Development*, 1989.

Defining Object-Level Parsers in λProlog
Extended Abstract

Amy Felty
AT&T Bell Laboratories
600 Mountain Ave.
Murray Hill, NJ 07974 USA
felty@research.att.com

1 Introduction

The higher-order logic programming language λProlog contains the simply-typed λ-terms as its basic data structures. These terms can be used to elegantly express the ***higher-order abstract syntax*** [12, 8] of objects that include notions of bound variables such as formulas, proofs, and programs. Current implementations of λProlog, however, have no provision for a programmer to provide a *concrete syntax* for a particular object-level language. Such a capability is desirable, for example, in implementing an interactive theorem prover. Providing the user with a familiar syntax for the logic being implemented can greatly enhance interaction.

In this abstract, we propose an approach to providing programmer-defined concrete syntax. A simple *grammar specification language* will be used to describe grammar rules that translate the programmer's object-level concrete syntax to λProlog syntax. On the left hand side of each grammar rule, we include a term describing how to build the abstract syntax for the rule as a whole from the components on the right hand side. These terms represent an intermediate form approximating the higher-order syntax. They can be viewed as untyped λ-terms, extended to handle occurrences of both bound and free (logic) variables that are encountered in the object-level input.

From a grammar specification, we want to automatically generate a parser for an object language that can then be accessed by the λProlog programmer. There are many ways to generate such a parser. For illustration purposes, we will describe a technique using the Yacc parser generator [5] that was used in performing some initial experiments using the experimental Standard ML implementation (LP-SML) [2]. The implementation described here will generate parsers that use a two-step approach to parsing where the first step translates concrete syntax to an intermediate syntax which corresponds to the usual notion of parse trees, also called *first-order abstract syntax*. The second step, which translates first-order to higher-order abstract syntax will be presented as a λProlog program. Although a one phase approach implemented directly in ML may be more efficient, presenting the second phase as a λProlog program plays two roles. First, it provides a clear specification for what needs to be implemented in any one-phase approach, making operations such as those needed to handle variables and constants explicit. Second, it illustrates the use of λ-terms for expressing and manipulating higher-order abstract syntax in λProlog.

To illustrate the grammar specification language and its implementation, we will use a simple object language as an example throughout this paper. Our object language will be first-order

formulas. In the next section, we discuss higher-order syntax and introduce constants for expressing the higher-order syntax of our first-order object language. These constants are used to build the terms that are manipulated internally by a λProlog program, for example an interactive theorem prover for first-order logic. We then define a concrete syntax for such formulas that will be used by a user interacting with such a theorem prover. Then, in Section 3, we present the grammar specification language. Since part of the implementation will be described via a λProlog program, we describe this language and an interpreter for it in Section 4. In Section 5, we discuss the implementation of parsers from grammar specifications, and present the non-logical primitives added to λProlog to incorporate parsers. In Section 6, we present the λProlog program for the second phase of parsing, and in Section 7 we conclude.

2 Abstract Syntax in λProlog

The terms of λProlog are essentially those of the simply typed λ-calculus. We assume a fixed set of *primitive types*. *Function types* are constructed using the binary infix symbol `->`; if τ and σ are types, then so is τ `->` σ. The type constructor `->` associates to the right. If τ_0 is a primitive type then the type τ_1 `->` $\cdots$ `->` τ_n `->` τ_0 has $\tau_1, \ldots, \tau_n$ as *argument types* and τ_0 as *target type*. For each type τ, we assume that there are denumerably many constants and variables of that type. Simply typed λ-terms are built in the usual way using constants, variables, applications, and abstractions. Equality between λ-terms is taken to mean $\beta\eta$-convertibility. We shall assume that the reader is familiar with the usual notions and properties of substitution and α, β, and η conversion for the simply typed λ-calculus. See [4] for a fuller discussion of these basic properties.

In this paper, we adopt the syntax of the LP-SML implementation of λProlog. Free variables are represented by tokens with an upper case initial letter and constants are represented by tokens with a lower case initial letter. Bound variables can begin with either an upper or lower case letter. λ-abstraction is represented using backslash as an infix symbol. Terms are most accurately thought of as being representatives of $\beta\eta$-conversion equivalence classes of terms. For example, the terms `X\(f X)`, `Y\(f Y)`, `(F\Y\(F Y) f)` and `f` all represent the same class of terms.

Primitive types are introduced using *kind declarations* and constants are introduced using *type declarations*. For example, the following declarations introduce a new type and a binary functional constant.

```
kind    i       type.
type    f       i -> i -> i.
```

To represent a first-order logic, we introduce two primitive types: `form` for object-level formulas and `tm` for first-order terms. We then introduce constants for the object-level connectives as follows.

```
kind    tm      type.
kind    form    type.

type    and     form -> form -> form.
type    or      form -> form -> form.
```

```
type    imp     form -> form -> form.
type    neg     form -> form.
type    forall  (tm -> form) -> form.
type    exists  (tm -> form) -> form.
type    false   form.
```

By declaring **forall** and **exists** to take functional arguments, we have defined object-level binding of variables by quantifiers in terms of λ-abstraction, the meta-level binding operator. Thus, bound variables of the object language are identified with bound variables of the metalanguage of type `tm`. This representation of formulas was first introduced by Church [1]. We can also introduce constants at the meta-level to represent constants, function symbols, propositions, and predicates of first-order logic. For example, for a logic containing a constant c, a unary function symbol f, a unary predicate p, a binary predicate q, and a proposition r, we give the following declarations.

```
type    c       tm.
type    f       tm -> tm.
type    p       tm -> form.
type    q       tm -> tm -> form.
type    r       form.
```

Using these definitions, the first-order formula $\forall x(p(f(x)) \supset q(c,z))$, for example, is represented by the λ-term:

```
(forall X\ (imp (p (f X)) (q c Z)))
```

In our example, we will assume that a user interacts with a program such as a theorem prover using a more familiar concrete syntax that will be transformed internally to the above syntax. The concrete syntax we adopt here will closely resemble the usual syntax. In particular, we replace the commonly used symbols $\wedge$, $\vee$, $\supset$, $\neg$, $\forall$, $\exists$, $\bot$ with the ascii strings **&**, **or**, **=>**, **not**, **all**, **some**, **false**, respectively. In addition, we will use a dot after a quantifier and bound variable, which may sometimes replace the parentheses around the quantified expression. For instance, when entering the formula in the example above to a theorem prover, a user would type:

```
all x. p(f(x)) => q(c,Z)
```

In our grammar specification language, we must include a provision for specifying the class of atoms of the metalanguage to which a particular identifier of the concrete syntax may belong. We choose to allow concrete syntax to contain any one of the three atomic expressions in the metalanguage: constants, free (logic) variables, or bound variables. As we will see, bound variables will be treated the same as constants, and thus we will have only two categories of atoms. During parsing of a particular input, the class to which each individual identifier belongs is determined. For our first-order logic, we will restrict occurrences of function symbols, predicates, and propositions to be constants, while atomic first-order terms can be either constants or variables. In the example above, `x` in the expression `f(x)` will be parsed to a bound variable occurrence, `Z` to a free variable, and `c` to a constant, while the predicates `p` and `q` and the function symbol `f` all correspond to constants of the metalanguage.

⟨*left*⟩	::=	(⟨*class-name*⟩ ⟨*l-term*⟩)
⟨*l-term*⟩	::=	(⟨*atom*⟩ ⟨*string*⟩)
	|	(⟨*atom*⟩ ⟨*ident*⟩)
	|	⟨*ident*⟩
	|	(`app` ⟨*l-term*⟩ ⟨*l-term*⟩)
	|	(`abs` ⟨*ident*⟩ ⟨*l-term*⟩)
⟨*atom*⟩	::=	`const` | `var` | `cv`
⟨*r-elem*⟩	::=	(⟨*class-name*⟩ ⟨*ident*⟩)
	|	(⟨*lex-name*⟩ ⟨*string*⟩)
	|	(⟨*lex-name*⟩ ⟨*ident*⟩)

Figure 1: Grammar for Specifying Parsers

3 A Language for Specifying Parsers

A grammar for a particular object language is specified as a set of rules of the form ⟨*left*⟩ `-->` ⟨*right*⟩. Figure 1 specifies the form that the left and right sides of each grammar rule must take. The left hand side must be a ⟨*class-name*⟩ followed by an ⟨*l-term*⟩ which gives the form of the abstract syntax. This abstract syntax tree is built from the individual components found on the right hand side. The right side of a grammar rule is a list of elements described by ⟨*r-elem*⟩ above. An ⟨*r-elem*⟩ has one of three forms. If it is of the first form shown in the figure, the ⟨*class-name*⟩ indicates that the rules for the appropriate class must be used to parse the next token(s) from the input to obtain an item of this class. If successful, the term obtained will "instantiate" the identifier ⟨*ident*⟩ following ⟨*class-name*⟩. The remaining two forms handle literals or tokens in the input stream. A ⟨*lex-name*⟩ identifies a class of objects from the lexical analyzer. We do not go into detail about the lexical analysis phase in this paper, but just note that ⟨*lex-name*⟩ is provided to handle the interface between this phase and the parsing phase. We could simplify this interface by just allowing one ⟨*lex-name*⟩ called `literal` or `token`, for example. In our example we will have two such classes so that we may distinguish between symbols and identifiers. When the argument following ⟨*lex-name*⟩ is a string, the input must match the string exactly. The strings `"all"` and `"."` in the syntax of universally quantified formulas will be examples of such tokens occurring in our grammar for first-order logic. When the argument following ⟨*lex-name*⟩ is an identifier, the next token in the input stream will instantiate this identifier as long as it is from the class specified by ⟨*lex-name*⟩.

The terms representing the abstract syntax tree have the form specified by the ⟨*l-term*⟩ grammar. They can be viewed as untyped λ-terms extended with constructors used to indicate classes for atoms. Expressions for atoms are specified by the first two clauses of the grammar. The constants `const` and `var` take as arguments objects that will correspond to constants or variables, respectively, of the metalanguage. The keyword `cv` takes an argument that is permitted to be either a constant or variable. These constants and variables are represented either by a string or an identifier. A string specifies a specific constant or variable. An ⟨*l-term*⟩ can also be simply an identifier or can be an application or abstraction built using `app` and `abs`. Any identifier occurring in an ⟨*l-term*⟩ on the left of a grammar rule must also appear on the right in an ⟨*r-elem*⟩.

```
(formula A) --> (form_imp A)
(form_imp (app (app (const "imp") A) B)) --> (form_and A) (symbol "=>")
                                               (form_imp B)
(form_imp A) --> (form_and A)
(form_and (app (app (const "and") A) B)) --> (form_and A) (symbol "&")
                                               (form_atom B)
(form_and A) --> (form_atom A)
(form_atom (app (const "forall") (abs X A))) --> (symbol "all") (ident X)
                                                   (symbol ".") (formula A)
(form_atom A) --> (symbol "(") (formula A) (symbol ")")
(form_atom (const A)) --> (ident A)
(form_atom A) --> (pre_ap A) (symbol ")")
(pre_ap (app (const P) M)) --> (ident P) (symbol "(") (term M)
(pre_ap (app P M)) --> (pre_ap P) (symbol ",") (term M)
(term M) --> (symbol "(") (term M) (symbol ")")
(term (cv M)) --> (ident M)
(term M) --> (pre_ap M) (symbol ")")
```

Figure 2: A Grammar for First-Order Logic

Figure 2 contains a grammar specification of a parser for our first-order logic using this language. We only consider conjunction, implication, and universal quantification here. The other connectives are handled similarly. This grammar illustrates how precedence and associativity can be handled in this framework. Here, conjunction binds tighter than implication, and implication is right-associative, while conjunction is left-associative. Each of the constants `symbol` and `ident` appearing on the right hand side in the rules is a lexical class (or ⟨*lex-name*⟩) for symbols and identifiers, respectively. These two classes are defined as regular expressions in the lexical analyzer. We do not give their specifications here. There are four classes for formulas in the grammar. The first is for the general category of formulas and is defined by the first rule in the figure: `A` is a formula if `A` belongs to the `form_imp` class. This latter class handles implications and its associativity. The first of the two rules for this class state that a formula is an implication if it has a formula with no top-level implication on the left, and a formula possibly with a top-level implication on the right. In the abstract syntax term on the left, an implication is represented as the constant `imp` applied to its two formula arguments. The keyword `const` is used in this term to indicate that its argument corresponds to a constant of the metalanguage. The second rule for `form_imp` handles the case when there is no top-level implication. Formulas with no top-level implication are described by the `form_and` class. The fact that this class is a subclass of `form_imp` insures that implication does not bind as tightly as conjunction. This class is similar to `form_imp` except that the associativity is reversed. In this case, `form_atom` is the subclass for formulas with no top-level `&` or `=>`. A `form_atom` is either universally quantified or is an atomic formula. The first rule for this class handles universal quantification. In a particular instance, the identifier that is assigned

to `X` may have occurrences in the structure assigned to `A`. The fact that these occurrences should be considered bound is recorded on the left by using the `abs` construct. In the second rule, we allow atomic formulas to be parenthesized. The third rule handles propositions. Since we stated in the previous section that we restrict propositions to be meta-level constants, the keyword `const` is used on the left side. The fourth and last rule of this class handles predicates applied to one or more arguments, as defined by the `pre_ap` class, and is terminated by a right parenthesis. A member of the `pre_ap` class can either be a predicate symbol followed by a left parenthesis and a member of the `term` class for terms of our first-order logic, or a `pre_ap` followed by a comma and then a `term`. A term as specified by the `term` class can occur inside parentheses, it can be atomic, in which case it may correspond to a constant or variable of the metalanguage, or it can be a function symbol applied to one or more arguments. The `pre_ap` class handles the third case in the same way that it handles predicates. The keyword `const` is used on the right of the first rule of the `pre_app` class since both function symbols and predicates must be constants.

4 λProlog

Formulas are introduced into λProlog by including a primitive type `o` for propositions, and introducing suitable constants with their types for the logical conectives and quantifiers. In particular, we introduce constants for conjunction (`,`), disjunctions (`;`), and implication (`=>`) having type `o -> o -> o`. The constants for universal quantification (`pi`) and existential quantification (`sigma`) are given type `(A -> o) -> o` for each type replacing the "type variable" `A`. A function symbol whose target type is `o`, other than a logical constant, will be considered a *predicate*. A λ-term of type `o` such that the head of its $\beta\eta$-long form is not a logical constant will be called an *atomic formula*.

We define two classes of propositions, called *goal formulas* and *definite clauses* (or just *clauses*). Let A be a syntactic variable for atomic formulas, G a syntactic variable for goal formulas, and D a syntactic variable for definite clauses. These two classes of formulas are defined by the following mutual recursion.

$$G := A \mid G_1\texttt{,}G_2 \mid G_1\texttt{;}G_2 \mid \texttt{sigma}\ x\backslash G \mid \texttt{pi}\ x\backslash G \mid D\texttt{=>}G$$

$$D := A \mid \texttt{pi}\ x\backslash D \mid G\texttt{=>}A$$

A *logic program* is a finite set of definite clauses. When we write definite clauses, we will omit outermost universal quantifiers. In addition, the outermost implication, if there is one, will be written using `:-` which denotes the converse of implication. In a definite clause of the form $A\texttt{:-}G$, the atomic formula A is called the *head* of the clause, and G is called the *body*. There is one final restriction on definite clauses: the head of a definite clause must have a constant as its head. The heads of atomic goal formulas on the other hand may be either variable or constant.

A complete non-deterministic search procedure based on intuitionistic provability can be defined by the following six search operations [9]. In these operations, $\mathcal{P}$ is the current program and G is the current goal.

AND: If G is $(G_1\texttt{,}G_2)$ then try to show that both G_1 and G_2 follow from $\mathcal{P}$.

OR: If G is (G_1;G_2) then try to show that either G_1 or G_2 follows from $\mathcal{P}$.

INSTANCE: If G is (`sigma` $x\backslash G'$) then try to show that there is some term t of the same type as x such that $[t/x]G'$ is provable from $\mathcal{P}$.

GENERIC: If G has the form (`pi` $x\backslash G'$) then pick a new parameter c and try to prove $[c/x]G'$ from $\mathcal{P}$.

AUGMENT: If G has the form (D=>G') then proceed to attempt to prove G' from $\mathcal{P} \cup \{D\}$.

BACKCHAIN: If G is atomic, we consider the current program. If there is a universal instance of a program clause which is equal to G then we have found a proof. If there is a program clause with a universal instance of the form G:-G' then try to prove G' from $\mathcal{P}$.

The λProlog interpreter makes choices which are left unspecified by the high-level description of the non-deterministic interpreter, many of which are similar to those routinely used in Prolog. The order in which conjuncts and disjuncts are attempted and the order for backchaining over definite clauses is determined exactly as in conventional Prolog: conjuncts and disjuncts are attempted in the order they are presented. Definite clauses are backchained over in the order they are listed in $\mathcal{P}$ using a depth-first search paradigm to handle failures. In the extended language, clauses can be added dynamically by the AUGMENT operation. We specify that new clauses get added to the top of the list.

In the INSTANCE operation, the Prolog implementation technique of instantiating the existential quantifier with a logic (free) variable which is later "filled in" using unification is employed. Thus instead of picking a term t, the INSTANCE search operation will introduce a new logic variable as the substitution term. A similar use of logic variables is made in implementing BACKCHAIN: a clause from $\mathcal{P}$ is chosen and an instance is made by replacing all outermost universally quantified variables with new logic variables. This universal instance of the clause is then unified with the current goal. This operation may partially or fully instantiate the new logic variables. The addition of logic variables in our setting requires higher-order unification since these variables can occur inside λ-terms.

The presence of logic variables requires that GENERIC be implemented slightly differently than is described above. In particular, if the goal or the current program $\mathcal{P}$ contains logic variables, the new constant introduced by this operation must not appear in the terms eventually instantiated for those logic variables.

λProlog permits a degree of polymorphism by allowing type declarations to contain type variables (written as capital letters). We will make use of this polymorphism in our program for translating first-order to higher-order syntax. This program will be used to translate objects of arbitrary type.

5 λProlog Primitives for Parsing

In generating parsers from grammar specifications, it is possible to employ one of the well-studied parsing methods or use existing parser generator tools [13, 7, 5, 6, 3, 11]. Choosing to use a

particular method will have impact on what kinds of grammars may be accepted as well as on efficiency of the resulting parser. As an example, we choose the Yacc parser generator [5], and thus implement LALR grammars. We use the ML-Lex and ML-Yacc tools to implement lexical analyzers and parsers, respectively. These tools are the ML versions of the unix lex and yacc utilities.

The first phase of the procedure uses the ML-Lex tool for generating lexical analyzers which transform an input stream to a list of tokens. We will say very little about this phase here. In LP-SML, a lexical analyzer for a user-defined object language can be derived in a straightforward way from the ML-Lex specification for λProlog syntax. In our example, we choose to parse identifiers in the same way that λProlog does, so we take this information directly from the existing ML-Lex specification. We must then add rules for the literal strings representing the connectives of first-order logic.

Any specification in our grammar specification language can be transformed in a straightforward manner to input to ML-Yacc. We view the constants `app`, `abs`, etc., as constructors for λProlog terms representing a first-order approximation of the desired higher-order syntax. The ML-Yacc phase of parsing will build the internal representation of these λProlog terms. For the final phase of parsing, in the next section, we present a λProlog program that transforms this first-order syntax to higher-order syntax. Since λProlog terms are typed, we must make sure that a term obtained from a translation from concrete syntax is correctly typed. As we will see, type checking is handled by the final phase of parsing.

In order to accommodate user-defined parsers, we provide two new commands, `use_parser` used to generate and load a parser, and `parse` used to call the parser on particular expressions in an object language. They have the following types.

```
type  use_parser  string -> o.
type  parse       string -> string -> (A -> o) -> o.
```

The argument to `use_parser` is the name of the file containing the grammar specification. For example, if the grammar for first-order logic in Figure 2 were in a file called fol.gram, the command `(use_parser "fol")` will read in the file, create the specification of the lexical analyzer and use ML-lex to generate it, and create the Yacc specification and use ML-Yacc to generate the parser which translates concrete syntax to intermediate terms. A goal of the form `(parse Parser In G)` uses the parser named by `Parser` on the input `In`. In writing the interactive component of a program such as a theorem prover in λProlog, the programmer will make use of standard read and write predicates as in Prolog. Here, we assume that input can be obtained from read predicates in the form of a string which can then be passed on to the `parse` command. If the parse fails on the string `In`, the goal fails. Otherwise, an output term `Out` is obtained representing the higher-order abstract syntax of the input term, and then the goal `(G Out)` is attempted. Here, the type of `Out` is unified with the type of the bound variable in `G` and an error is signalled if this type-unification fails.

During execution of a λProlog program, new constants will be generated dynamically by the GENERIC operation and new logic variables will be generated by INSTANCE and BACKCHAIN. In an interactive session, we will want to make at least some of these constants and variables accessible to the user, so that they may be accepted as input by user-defined parsers. For the purposes of this

paper, we will assume that there is some method by which names are established for new constants and variables, and only those with established names can be accessed by the user. For example, one way in which logic variables can get established names is by being printed out to the screen by an output command. More specifically, if a term to be output to the screen contains a logic variable that does not already have a name, a name is chosen that does not conflict with the names of currently existing variables and is established for that variable.

Establishing the correspondence of the objects in the input stream to actual constants and variables with established names will take place during the second phase of parsing. To make this correspondence for variables, we will make use of a non-logical λProlog primitive `fvar` of type `A -> string -> o`. A goal of the form `(fvar V Name)` will succeed if `V` is a logic variable with the established name `Name`. It will also succeed if `V` is a variable with no established name. If there is some other variable `V'` with established name `Name`, `V` will be set equal to `V'`. Otherwise, the name `Name` will be established for `V`.

Logically, any variable found in a user's input that doesn't already exist with an established name can be viewed as a new one generated by INSTANCE. In a goal of the form `(parse Parser In G)`, if the resulting term `Out` has n new variables $X_1, \ldots, X_n$ that didn't already have established names, then consider the term `Out'` with bound variables $X_1, \ldots, X_n$ and body `Out`. Then, the goal we solve after a successful parse is actually:

$$\texttt{(Z\textbackslash(sigma } X_1\texttt{\textbackslash(}\ldots\texttt{(sigma } X_n\texttt{\textbackslash(G (Z } X_1 \ldots X_n\texttt{)))}\cdots\texttt{))) Out'}.$$

6 Translating First-Order to Higher-Order Syntax

We introduce the type `iterm` for the intermediate terms that are constructed by the Yacc-generated parser. The constants app, abs, etc., introduced in our grammar clauses will be considered constructors for terms of this type. They have the following types.

```
type  app        iterm -> iterm -> iterm.
type  abs        string -> iterm -> iterm.
type  const      string -> iterm.
type  var        string -> iterm.
type  cv         string -> iterm.
```

In addition, we have the following predicates that will be used in implementing the syntax translation.

```
type  nameof     A -> string -> o.
type  trans      iterm -> A -> o.
```

The `nameof` predicate handles the translation of constants and occurrences of bound variables in object-language terms. It relates a meta-level constant to the string containing its name. A type variable is used for the first argument since these constants can be of any type. Before translation

of a particular term, we start with one **nameof** clause for every constant in the environment with an established name. We also need to know the type of each constant, so we must include type declarations. Dynamically generated constants that have no established name need not be considered since they have no external representation visible to the user. Constants with established names include at least all those declared by the programmer. Thus, when parsing a formula of first-order logic, for example, we must include at least the following declarations and clauses.

```
kind  form       type.
kind  tm         type.

type  and        form -> form -> form.
type  imp        form -> form -> form.
type  forall     (tm -> form) -> form.
type  c          tm.
type  f          tm -> tm.
type  p          tm -> form.
type  q          tm -> tm -> form.
type  r          form.

nameof and "and".
nameof imp "imp".
nameof forall "forall".
nameof c "c".
nameof f "f".
nameof p "p".
nameof q "q".
nameof r "r".
```

A **nameof** clause will be added dynamically for each binding occurrence of a variable that is encountered during parsing. Then, as parsing proceeds, each argument to **const** or **cv** will be checked against the existing **nameof** pairs. For an argument to **const**, if it does not match anything, the parse fails. An argument to **cv**, if it is not a constant, will be interpreted as a free variable. The **trans** predicate used for the general translation takes two arguments. The first is the input. It is the result of the parse by the Yacc-generated parser, and thus is the intermediate first-order syntax. The second argument is the resulting term in the desired higher-order syntax. The translation is defined by the following clauses.

```
trans (app M N) (P Q) :- trans M P, trans N Q.
trans (abs X M) N :- pi c\ (nameof X c => trans M (N c)).
trans (const M) N :- nameof N M, !.
trans (var M) N :- fvar N M.
trans (cv M) N :- nameof N M, !.
trans (cv M) N :- freevar N M.
```

The first two clauses handle application and abstraction. In an application, each argument is translated and the result of the first translation is directly applied to the second. Note that types must match in order for this clause to succeed. `P` must have a functional type and `Q` must have the appropriate argument type. Otherwise, the translation fails. The clause for abstraction transforms an intermediate term with occurrences of string representation of a bound variable to a term of the metalanguage containing an actual abstraction. In this clause, the GENERIC operation is used to introduce a new constant, say `c`, to play the role of the bound variable. The AUGMENT operation adds the atomic clause relating the string representation of the bound variable to this constant. This clause is available while translating the body `M`. It will be used to replace all occurrences of the string `X` in the intermediate term `M` to the constant `c`. If successful, the result of the translation must match the template `(N c)`. `N` will be the term obtained by abstracting out all occurrences of `c`. It is important that the new clause added by AUGMENT be added to the top of the list of `nameof` clauses. If a bound variable is introduced with the same name as an existing constant, it is important that all occurrences within the scope of the bound variable get parsed as occurrences of this bound variable and not as the already existing constant.

The last four clauses pertain to translation of atoms. The non-logical feature cut (`!`) of λProlog is needed in these clauses. It is used to eliminate backtracking points. It is a goal which always succeeds and commits the interpreter to all choices made since the parent goal was unified with the head of the clause in which the cut occurs. Here, we do not want backtracking to cause an identifier to be interpreted as more than one kind of atom. The first clause uses `nameof` to translate constants or occurrences of bound variables. The next clause translates free variables using the `fvar` primitive to determine if the variable occurs in the current context, and to generate a new one when it doesn't. The result of the translation is the already existing or the new variable. The next two clauses handle an atom that can be either a constant or variable. The order in which they are attempted is important. First, it must be checked whether it occurs within the scope of a bound variable or is a constant. If not, it is a logic variable.

We end this section by discussing how the last phase of parsing fits in with the rest. As stated in the previous section, a goal of the form `(parse Parser In G)` uses the parser named by `Parser` on the input string `In`. It does so in three steps. First, it will run the lexical analyzer, and second, it will run the Yacc-generated parser on `In` to obtain a term, say `Mid`, the intermediate syntax representation of the input. Let `parser` be the name of a λProlog module containing all the code presented in this section except the clauses specific to the first-order logic example. Let `constants` be the module containing the type declarations and `nameof` clauses for all the constants with established names in the current environment. The final step of the `parse` command is an attempt to solve the following goal.

```
parser ==> (constants ==> ((trans Mid Out),(G Out)))
```

The `==>` symbol is the meta-level connective that instructs the interpreter to load the module named on the left of the arrow into memory and add all of the clauses in this module to the current program. Note that the `parser` module is a static object, while the `constants` module must be created dynamically since it will depend on the environment at the time the `parse` command is invoked.

7 Conclusion

We have proposed a high-level specification language for integrating object-language parsers into λProlog. For illustration purposes, we described a two phase method of implementing this facility. There are several other possibilities. For example, still using the ML-Yacc facility, a one phase approach can be implemented directly in ML and may be more efficient. To do so, instead of building the λProlog terms of type `iterm`, we can view the constants `app`, `abs`, etc., as ML functions which take their arguments and directly form the internal ML representation of the appropriate higher-order syntax. The operations handled by the `trans` program must now be handled by these functions. Thus, for example, these functions must distinguish between constants, bound variables, and free variables and keep track of their scope, recognize occurrences of free (logic) variables that exist in the current environment, add to the current environment any new logic variables that occur in a successfully parsed term, and verify that the resulting term is well-typed.

Another possibility is to consider a form of definite clause grammars as in Prolog [11]. In fact, the grammar specification in Figure 2 already has a form much like a definite clause grammar. In [10], an extension of definite clause grammars to handle scoping constructs is described. It would be straightforward to implement our grammar in the manner described in that paper. In doing so, we obtain a λProlog program to parse a list of tokens to a term of type `iterm`, where the left hand sides of rules in Figure 2 correspond to the heads of clauses and the right to the body. The list of elements on the right become a conjunction of subgoals. In fact, we can modify such a program so that it incorporates the `trans` program and performs parsing from a list of tokens to higher-order syntax in a single phase. However, a λProlog program obtained from this grammar cannot be executed directly. To see why, note that there is a rule for the `form_and` class where the first element on the right also requires a term from the `form_and` class. In the corresponding program, `form_and` will be a predicate and using the clause corresponding to this rule will cause infinite branching in the search. This is a common problem in a grammar with infix operators. It is possible to change the grammar to obtain an executable parser, though care must be taken in doing so.

In this paper, we have introduced a `parse` command to explicitly call a parser on a given input. At the point such a call is made, only the syntax of the given object language can be parsed. Object-level terms cannot contain arbitrary λProlog syntax inside them. In some cases, it may be desirable to mix the two syntaxes. For example, the programmer may want to write programs that use object-level syntax inside clauses, and may not want to have to invoke a `parse` command explicitly to do so. For instance, a user should at least be able to specify new infix symbols. To handle this, some method for integrating the existing λProlog parser with user-defined parsers will be needed.

Acknowledgments

The author would like to thank Elsa Gunter, Dale Miller, Olivier Nora, Fernando Pereira, and Konrad Slind for valuable discussions on this topic.

References

[1] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[2] Amy Felty, Elsa Gunter, and Frank Pfenning. LP-SML, a Standard ML Implementation of λProlog. In progress.

[3] J. Heering and P. Klint. The syntax definition formalism SDF. In J. A. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, chapter 6, pages 283–297. Addison-Wesley, 1989.

[4] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinatory Logic and Lambda Calculus.* Cambridge University Press, 1986.

[5] S. C. Johnson. Yacc—yet another compiler compiler. Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, NJ, 1975.

[6] G. Kahn, B. Lang, B. Mélèse, and E. Morcos. Metal: a formalism to specify formalisms. *Science of Computer Programming*, 3:151–188, 1983.

[7] Michel Mauny and Daniel de Rauglaudre. Parsers in ML. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, June 1992.

[8] Dale Miller. Abstract syntax and logic programming. In *Logic Programming: Proceedings of the First and Second Russian Conferences on Logic Programming*, number 592 in Lecture Notes in Artificial Intelligence, pages 322–337. Springer-Verlag, 1992.

[9] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[10] Remo Pareschi and Dale Miller. Extending definite clause grammars with scoping constructs. In D. H. D. Warren and P. Szeredi, editors, *International Conference in Logic Programming*, pages 373–389. MIT Press, June 1990.

[11] F.C.N. Pereira and D.H.D. Warren. Definite clause grammars for language analysis—a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13:231–278, 1980.

[12] Frank Pfenning and Conal Elliot. Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, 1988.

[13] C. Reade. *Elements of Functional Programming.* International Computer Science Series. Addison-Wesley, 1989.

A Deductive Database View of Embedded Implications

Burkhard Freitag
Technische Universität München
Institut für Informatik
Orleansstrasse 34, D-W8000 München 80
Germany
freitag@informatik.tu-muenchen.de

1 Abstract

In this paper a modular deductive database language based on embedded implications is presented. Our language can be considered a subset of λ-Prolog [23]. Its notion of embedded implication can best be compared to that of the module system of λ-Prolog. We allow negation-as-failure in subgoals of a rule or query as well as in the consequence of an embedded implication. The main motivation for the definition of our language has been the desire to make a notion of local definitions available for deductive database systems having a bottom-up query evaluation strategy, e.g. the *LOLA*-system [7].

```
Mapl  := { maplist([],[]).
           maplist([X|L],[Y|R]) :- f(X,Y), maplist(L,R). }

Conv  := { convert(In,Out) :- ( maplist(In,Out) <= Mapl ).
           f(U,V) :- look_up(U,V). }
```

Figure 1: Sample programs Mapl and Conv in basic syntax

2 Basic Language

A BNF-style grammar of our basic syntax is shown in figure 2. Rules are implicitly universally quantified. This applies also to the rules referenced by an implicational subgoal. Only programs, i.e. sets of *universally closed formulas*, are allowed in the antecedents of implicational subgoals. This is a major restriction[1] as compared to λ-Prolog [23] and the languages proposed e.g. in [19], [16], [12], [5]. Neither in rules nor in goals multiple consequences or nested implications are allowed. However, such programs and goals can easily be transformed into the basic syntax. Program names simply serve as placeholders for the corresponding rule sets. In particular, we do not allow mutual program references. Sample programs[2] in basic syntax are shown in figure 1.

[1] Work is underway to define a less restrictive syntax that still preserves bottom-up evaluability.

[2] To the sample programs shown in this paper in general a variant of the Magic Set Transformation [2], [3] has to be applied to generate safe, i.e. bottom-up evaluable, rules.

```
⟨query⟩              ::=  :- ⟨body⟩.
⟨program⟩            ::=  ⟨rule-set⟩
⟨rule-set⟩           ::=  { ⟨rule⟩* }
⟨rule⟩               ::=  ⟨head⟩.
                      |   ⟨head⟩ :- ⟨body⟩.
⟨body⟩               ::=  ⟨subgoal⟩ {, ⟨subgoal⟩ }*
⟨subgoal⟩            ::=  ⟨i-subgoal⟩
                      |   ⟨literal⟩
⟨i-subgoal⟩          ::=  ( ⟨literal⟩ <= ⟨reference⟩ {, ⟨reference⟩}* )
⟨reference⟩          ::=  ⟨program-reference⟩
⟨program-reference⟩  ::=  ⟨program-name⟩
                      |   ⟨program⟩
⟨literal⟩            ::=  ⟨atom⟩
                      |   $not ⟨atom⟩
⟨atom⟩               ::=  ⟨predicate-symbol⟩
                      |   ⟨predicate-symbol⟩ ( ⟨term⟩ {, ⟨term⟩}* )
⟨term⟩               ::=  ⟨variable⟩
                      |   ⟨function-symbol⟩
                      |   ⟨function-symbol⟩ ( ⟨term⟩ {, ⟨term⟩}* )
⟨predicate-symbol⟩   ::=  ⟨functor⟩
⟨function-symbol⟩    ::=  ⟨functor⟩
⟨functor⟩            ::=  { a - z } ⟨string⟩
⟨variable⟩           ::=  { A - Z | _ } ⟨string⟩
⟨program-name⟩       ::=  { A - Z } ⟨string⟩
⟨string⟩             :=   { a - z | A - Z | 0 - 9 | $ | _ }*
```

Optional parts are enclosed in the meta–symbols [...] and groups are enclosed in the meta–symbols {...}. Repetition of a group is indicated by {...}*. The meta–symbols should be distinguished from the four syntax elements `[`, `]`, `{`, and `}`, respectively.

Figure 2: BNF–style Grammar of Basic Syntax

3 Bottom-Up Evaluation

As opposed to λ–Prolog [23] and most other query languages with embedded implications proposed in the literature (e.g. [19], [4], [5]), our language does *not* have a Prolog–like top–down evaluation strategy.

Instead, a *top–down query compilation* into an evaluating relational expression is applied in analogy to the ordinary deductive database case (For more details see e.g. [26], [7]). In a subsequent *bottom–up query evaluation* phase, the proper set of answer tuples is computed in a set–at–a–time fashion. Our query evaluation scheme thus generalizes the evaluation scheme of most deductive database systems because embedded implications can be handled. Many researchers consider bottom–up evaluation superior over Prolog–like top–down evaluation if large quantities of data have to be processed, which will typically occur in database–like applications, e.g. in traffic information systems. In the presence of context extensions as introduced by embedded implications, a combination of resolution and context extension has to be applied in the top–down compilation step. Context

extension may precede the resolution step, if the predicate symbols are labelled by an appropriate context identifier, e.g. the lexicographically ordered list of names of the programs forming the context. In figure 3 the different program contexts encountered during processing the sample query `:- ( convert(In,Out) <= Conv ).` are made visible by labelling the predicate symbols. However, it is not necessary that the context extension step precedes the actual query compilation. The results presented in [9] indicate how to interleave labelling and compilation for deductive database programs with embedded implications. For more details see [8] and [9].

```
Conv|Conv| :=
{   convert|Conv|(In,Out) :- maplist|Conv,Mapl|(In,Out).
    f|Conv|(U,V) :- look_up|Conv|(U,V). }

Conv|Conv,Mapl| :=
{   convert|Conv,Mapl|(In,Out) :- maplist|Conv,Mapl|(In,Out).
    f|Conv,Mapl|(U,V) :- look_up|Conv,Mapl|(U,V). }

Mapl|Conv,Mapl| :=
{   maplist|Conv,Mapl|([],[]).
    maplist|Conv,Mapl|([X|L],[Y|R])   :-   f|Conv,Mapl|(X,Y),
                                           maplist|Conv,Mapl|(L,R). }
```

Figure 3: Program contexts made visible by labelling predicate symbols

4 Perfect Model Semantics

The operational semantics of programs in basic syntax as sketched above is an almost direct implementation of an iterated fixpoint semantics [1] or perfect model semantics [24], that has been defined for stratifiable programs with negation–as–failure and embedded implications in [8]: Following the line of [19] we define the generalized Herbrand interpretations and a validity relation $\Vdash$ between generalized interpretations and goals. The immediate consequence operator $T_{\mathcal{W}}$ of a set $\mathcal{W}$ of programs maps the set of generalized Herbrand interpretations onto itself. As a major difference to [19], we do not require at this point that generalized Herbrand interpretations are internally monotonic (see below). It can be shown that generalized interpretations in our slightly more general sense as well as the generalized immediate consequence operator have the essential model theoretic properties just as in the ordinary deductive database case (cf. [1]). Consequently, a minimal fixpoint of $T_{\mathcal{W}}$ is a minimal model with respect to $\Vdash$ of the set of programs $\mathcal{W}$.
A natural ordering is imposed on a set $\mathcal{W}$ of programs by their *implicational depth*, i.e. the length of reference chains to other programs[3]. Therefore we can *horizontally* partition a set of programs into a sequence of *i–strata* $\mathcal{W}_k$, i.e. sets of programs with equal implicational depth k. It can be observed that the elements of an i–stratum do not refer to each other and thus

[3]Note, that program names are allowed in the antecedents of embedded implications.

may be processed simulaneously. By stratifiability with respect to negation every single program in an i–stratum $\mathcal{W}_k$ can be partitioned into an ordered set of *n–strata.* We can think of the collection of n–strata as partitioning every $\mathcal{W}_k$ in *vertical direction.* By every such vertical partition a corresponding immediate consequence operator is defined. In analogy to the fixpoint procedure for ordinary deductive database programs, we compute a sequence of minimal fixpoints of the immediate consequence operators proceeding from the rightmost i–stratum $\mathcal{W}_0$ to the leftmost i–stratum, and, within each i–stratum $\mathcal{W}_k$, starting at the lowest n–strata and proceeding to the highest n–strata. Using the techniques of [1] it can be shown that a minimal generalized model of $\mathcal{W}$ is computed which, indeed, is the perfect generalized Herbrand model [24]. See [8] and [9] for more details.

5 Negation–As–Failure

Negation–as–failure is known to be problematic in ordinary deductive database programs due to its intrinsic nonmonotonic behaviour. On the other hand, deductive database programs often have to rely on implicit negative information, i.e. a form of negation–as–failure, because there are situations in which one simply does not have explicit negative information available. For instance, one would like to avoid to explicitly state an inequality axiom for every pair of constants introduced by base relations[4].

The situation grows more difficult if rules or queries have implicational subgoals. Sets of programs have to be processed simultaneously to account for references to other programs (cf. [6]). If we allow an unrestricted use of negation–as–failure the generalized Herbrand interpretations generated by the immediate consequence operators are not in general internally monotonic (cf. [19]). From the more procedural point of view it might appear quite natural to get a smaller set of true formulas when more information becomes available in an extended context. However, from a logical point of view the use of negation–as–failure should be restricted in a way that the internal monotonicity of the generalized perfect model is preserved. An extension of our syntax and semantics allowing to quantify over free variables of a module is currently under investigation. In many cases, this extension should make it possible to shift negation–as–failure down to the base relations where its use can be controlled.

As a more logical justification of negation–as–failure we refer the reader to the literature on circumscription (e.g. [15], [17], [14]) which has been used since long by the AI community to formalize nonmonotonic reasoning.

6 Modules and Static Scoping

Our semantics of embedded implications induces a *dynamic scoping rule* for predicates. While this behavior is suitable for hypothetical reasoning, it is clear that from a software engineering point of view *static scoping* should be preferred [18], [21]. Consequently, the basic syntax of our language is

---

[4] See e.g. the definition of the `not_member`-predicate in figure 5 and its use in the definition of `path`.

```
⟨module⟩                    ::=     $module ⟨module-name⟩ :
                            ::=     ⟨module-interface⟩ ⟨module-implementation⟩
⟨module-interface⟩          ::=        [ ⟨import-declarations⟩ ]
                                       [ ⟨export-declarations⟩ ]
⟨module-implementation⟩     ::=        [ ⟨local-declarations⟩ ]
                                       [ ⟨module-imports⟩ ]
                                       ⟨ rule-set ⟩
⟨import-declarations⟩       ::=     $import
                                       { ⟨declaration⟩ }*
⟨export-declarations⟩       ::=     $export
                                       { ⟨declaration⟩ }*
⟨local-declarations⟩        ::=     $local
                                       { ⟨declaration⟩ }*
⟨module-imports⟩            ::=     $import_modules
                                       { <= ⟨module-reference⟩ . }*
⟨declaration⟩               ::=     ⟨ predicate-schema ⟩.
⟨predicate-schema⟩          ::=     ⟨predicate-symbol⟩ ( { < ⟨functor⟩ > }* )
⟨module-reference⟩          ::=     ⟨output-arguments⟩ ⟨module-name⟩ ⟨input-arguments⟩
⟨output-argument-list⟩      ::=     ⟨argument-list⟩
⟨input-argument-list⟩       ::=     ⟨argument-list⟩
⟨argument-list⟩             ::=     []
                            ::=  |  [ ⟨argument-spec⟩ { , ⟨argument-spec⟩ }* ]
⟨argument-spec⟩             ::=     ⟨predicate-symbol⟩
                                 |  ⟨predicate-symbol⟩:=⟨predicate-symbol⟩
⟨module-name⟩               ::=     ⟨program-name⟩

...
Rules for basic syntax modified as follows
⟨reference⟩                 ::=     ⟨module-reference⟩
                                 |  ⟨program-reference⟩
```

Figure 4: BNF-style Grammar of Module Syntax

extended by a notion of module parameterization and predicate encapsulation. A BNF grammar of the module syntax is shown in figure 4. Sample modules can be found in figure 6 and figure 5.

A *module* consists of an *interface* and an *implementation*. In the interface, the import and export predicate symbols have to be declared[5]. The implementation part starts with declarations of the local predicate symbols and a list of references to the imported modules. By importing a module, the definitions of (some of) its exported predicates are made visible in the importing module. Consider the following declaration occurring in module `Graphs` of figure 5.

```
$import_modules
    <= [len:=length,nmemb:=not_member]Lists[].
```

[5]Currently, only the arity of predicate symbols is declared. The symbols occurring in a predicate schema are dummy attribute names. It is planned, however, to extend declarations to type declarations, e.g. as proposed in [22].

```
$module Graphs:
  $import
    ::= edge(<node>,<node>).
  $export
    ::= connected(<node>,<node>).
    ::= path(<node>,<node>,<list_of_nodes>).
  $local
    ::= nmemb(<item>,<list_of_items>).
    ::= len(<list_of_items>,<peano_integer>).
  $import_modules
    <= [len:=length,nmemb:=not_member]Lists[].
  {
    connected(X,Y)  :- edge(X,Y).
    connected(X,Y)  :- edge(X,Z), connected(Z,Y).
    path(X,Y,[X,Y]) :- edge(X,Y).
    path(X,Y,[X|P]) :- edge(X,Z), path(Z,Y,P), nmemb(X,P).
  }

$module Lists:
   $export
     ::= append(<list_of_items>,<list_of_items>,<list_of_items>).
     ::= member(<item>,<list_of_items>).
     ::= not_member(<item>,<list_of_items>).
     ::= length(<list>,<peano_integer>).
   $local
     ::= equal(<item>,<item>).
   {
     append([],L,L).
     append([X|L1], L2, [X|L3]) :- append(L1,L2,L3).
     member(X,[X|L]).
     member(X,[Y|L]) :- member(X,L).
     not_member(X,[]).
     not_member(X,[Y|L]) :- $not equal(X,Y), not_member(X,L).
     length([],0).
     length([X|L],s(N)) :- length(L,N).
     equal(X,X).
   }
```

Figure 5: Modules `Graphs` and `Lists`

The programmer states that the predicate symbol `length` of module `Lists` shall be visible within `Graphs` as the predicate `len`, and that `not_member` shall be visible as `nmemb`. Semantically, module import can be understood as a default module reference which is automatically added to the (possibly empty) premise of every subgoal occurring in a rule of the importing module [19]. The

```
$module Mapl:
   $import
     ::= f(Items,Items).
   $export
     ::= maplist(list(Items),list(Items)).
   {
     maplist([],[]).
     maplist([X|L],[Y|R]) :- f(X,Y), maplist(L,R).
   }

$module Conv:
   $import
     ::= look_up(Items,Items).
   $export
     ::= convert(list(Items),list(Items)).
   $local
     ::= map(list(Items),list(Items)).
   {
     convert(In,Out) :- (map(In,Out) <= [map:=maplist]Mapl[f:=look_up]).
   }

$module Table:
   $export
     ::= look_up(peano_integer,number).
   { look_up(0,0).
     look_up(s(0),1).
     ...  }
```

Figure 6: Sample modules Mapl, Conv, and Table

first connected–rule of module Graphs, for instance, is transformed into[6]

```
connected(X,Y) :- (edge <= {len(U,V) :-length(U,V).
                            nmemb(U,V) :- not_member(U,V).},
                           Lists_Rules)
```

where Lists_Rules denotes the rules of module Lists.
The local declarations are followed by a set of rules defining the exported and the local predicates. Imported predicates must not be defined within the importing module. In the module syntax, an implicational subgoal may have *module references* in its premise. A module reference is a program name surrounded by an *input* and an *output argument list* of the form $[p_1 := q_1, \ldots, p_1 := q_1]$, i.e. a list consisting of *argument specifications* $p_i := q_i$ where p_i and q_i are predicate symbols.

The following scoping and parameterization rules apply to our module language:

[6]In addition, every symbol of a module is labelled to provide for encapsulation. See also the section on unique module labelling below.

- Predicate symbols, regardless whether imported, exported, or local, are in general invisible outside the module, in which they are declared.
- Module parameterization is governed by the argument specifications occurring in the argument lists of a module reference. Only by an input (output) argument specification can the imported (exported) predicate symbols of a module be accessed. Access is realized by automatically generated linking rules (see below).

  Syntactically, the predicate symbol, that is closer to the module name, is the module parameter, and the symbol, that is closer to the enclosing module or query, is the actual argument symbol. Local predicate symbols can not be accessed at all from outside the module.

  In the module reference `[map:=maplist]Mapl[f:=look_up]`, for instance, the predicate symbol `look_up` is the actual argument for the input parameter predicate `f`, and `map` is the output predicate symbol serving as an actual argument of the output parameter predicate `maplist` (see figure 6). As for imported modules, embedded implications with module references are transformed into an embedded implication in basic syntax. The `convert`–rule of module `Conv` shown in figure 6, for instance, is transformed into

  ```
  convert(In,Out) :- ( map(In,Out) <= {map(X,Y) :- maplist(X,Y).
                                        f(X,Y) :- look_up(X,Y).},
                                       Mapl_Rules ).
  ```

  where `Mapl_Rules` denotes the set of rules of module `Mapl`. Note, that these are already in basic syntax.

  If a parameter predicate symbol p is literally the same as its actual argument, the corresponding argument specification p:=p may be abbreviated to p.
- The enclosing module[7] is the scope of all symbols either used as actual arguments in module references, or occurring in the conclusion of an implicational subgoal, or occuring in unnamed rule sets, that are part of the premise of an implicational subgoal. These symbols *must be declared* as local predicate symbols of the enclosing module.
- Program names without argument lists occurring in the premise of an implicational subgoal are treated like unnamed rule sets[8].

By a simple transformation the appropriate module instances can be obtained *at preprocessing or compilation time.* To this end, for each reference to a program a unique label is generated from the program name. The predicate symbols occurring in the program are subsequently prefixed by the so obtained unique symbol. Parameter passing is provided through special linking rules which the transformation generates from the user–defined input and output argument lists of a module reference. The labelled and transformed queries and modules are in the basic syntax and can be processed accordingly. For a more detailed description of the module syntax see [10] and [11].

[7] By convention, the enclosing module of a top-level query is the (empty) dummy module Top.

[8] Note, that unnamed rule sets and program names without argument lists are dynamically scoped within the enclosing module. By this feature hypothetical reasoning can be realized.

Assume, for example, that to an occurrence of the Conv–module has been assigned the label O. Then every symbol of this module occurrence is labelled by O, and every module reference occurring in Conv is assigned a new label, say 1. This process is continued until no new module reference is found[9]. After labelling and transformation into basic syntax, the convert–rule of the current module Conv reads as

```
convert_O(In,Out) :- ( map_O(In,Out) <= {map_O(X,Y) :- maplist_1(X,Y).
                                          f_1(X,Y) :- look_up_O(X,Y).},
                                         Mapl_Rules_1 ).
```

where Mapl_1 denotes the set of rules of module Mapl after labelling each symbol by the label 1. In this example no further labelling is required since Mapl does not contain module references.

In a language with higher order quantification the desired closure properties of a module M can be described by the formula ([13],[20]) $\forall in_1 \ldots \forall in_m \exists out_1 \ldots \exists out_n M$, expressing that the input predicates $in_1, \ldots, in_m$ are to be treated as formal parameters and, furthermore, that the output predicates $out_1, \ldots, out_n$ may depend on the $in_1, \ldots, in_m$. If the existential quantification is replaced by Skolem functions the higher order unification as introduced e.g. in [13] and [20]) gives the desired result. Our renaming transformation has essentially the same effect.

By the above scoping rules it should not be possible to access local predicates from outside a module. This can be achieved by shifting the rules defining local predicates into the premises of the body literals of the rules defining exported predicates as proposed in [19].

If modules are separately compiled into relation valued functions, we do not need the above described transformations. Instead, an appropriate parameterization is chosen for the generated evaluating functions. However, the transformation approach as well as the higher order unification approach show, that static scoping can be achieved without deviating very much from the pure logic language with embedded implications.

7 Implementation

A prototype system based on a preprocessor, that performs context extension by a source–to–source transformation has been implemented on top of the experimental deductive database system *LOLA*[10] developed at the Technische Universität München (TUM) [7].

8 Future Work

Currently, work is underway to define an appropriate form of negation–as–failure. In addition, we investigate the incremental compilation of modules and the combination of functional and logic programming obtained this way. Another direction of future research is the declarative formulation

[9]Note, that cyclic module references are not allowed.

[10]The *LOLA* project is a subproject of the joint effort "Objektbanken für Experten" between several german universities. It is funded by the German governmental institution "Deutsche Forschungsgemeinschaft" (DFG) under contract Ba 722/3–3 "Effiziente Verfahren zur logischen Deduktion über Objektbanken".

of constraints controlling the configuration of modules, which could be based on the notion of a module's signature as proposed e.g. in [25] .

References

[1] K.R. Apt, H.A. Blair, and A. Walker. Towards a Theory of Declarative Knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming.* Morgan Kaufmann, 1988.

[2] C. Beeri and R. Ramakrishnan. On the Power of Magic. In *Proc. of the 6th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems, San Diego*, pp. 269–283, 1987.

[3] I. Balbin, G. S. Port, and K. Ramamohanarao. Magic Set Computation for Stratified Databases. University of Melbourne, Dep. of Computer Science, Technical Report 87/3, Parkville, Australia, 1987

[4] A. Brogi, E. Lamma, and P. Mello. Inheritance and Hypothetical Reasoning in Logic Programming. In *Proc. of the 9th European Conf. on Artificial Intelligence*, 1990

[5] D. M. Gabbay and U. Reyle. N-PROLOG: An Extension of PROLOG with Hypothetical Implications. I.. *Journal of Logic Programming*, 319–355, 1984.

[6] D. M. Gabbay. N-PROLOG: An Extension of PROLOG with Hypothetical Implications. II.Logical Foundations and Negation as Failure. *Journal of Logic Programming*, 251–283, 1985.

[7] B. Freitag, H. Schütz, and G. Specht. *LOLA* - A Logic Language for Deductive Databases and its Implementation. In *Proc. 2nd Intl. Symp. on Database Systems for Advanced Applications (DASFAA '91), Tokyo, Japan*, 1991

[8] B. Freitag. Module und Hypothetisches Schliessen in Deduktiven Datenbanken. Technische Universität München, Ph.D. dissertation, in German, 1991.

[9] B. Freitag. Extending Deductive Database Languages by Embedded Implications. In A. Voronkov, editor, *Proc. Intl. Conf. on Logic Programming and Automated Reasoning (LPAR'92), St. Petersburg, July 1992*, LNAI 624, pp. 84 – 95. Springer–Verlag, 1992.

[10] B. Freitag. A Deductive Database Language Supporting Modules. In *Proc. 2nd Intl. Computer Science Conference (ICSC'92) – Data and Knowledge Engineering: Theory and Applications. Hong Kong, December 1992.*

[11] B. Freitag. A Deductive Database Language Supporting Modules. Universität Passau, Technical Report, MIP-9204, 1992

[12] L. Giordano, A. Martelli, and G. Rossi. Local Definitions with Static Scope Rules in Logic Programming. In *Proc. Intl. Conf. on Fifth Generation Computer Systems, 1988.*

[13] J. Hodas and D. Miller. Representing Objects in a Logic Programming Language with Scoping Constructs. In D. Warren and P. Szeredi, editors, *Logic Programming, Proc. 7th Intl. Conf. on Logic Programming.* MIT Press, 1990.

[14] V. Lifschitz. Computing circumscription. In *Proc. 9th Intl. Joint Conf. on Artificial Intelligence*, 1985.

[15] J. McCarthy. Circumscription: A form of non-monotonic reasoning. *Artificial Intelligence*, 13, 27–39, 1980.

[16] L. T. McCarty. Clausal intuitionistic logic. I. Fixed-point semantics. *Journal of Logic Programming*, 93–132, 1988.

[17] L. T. McCarty. Circumscribing embedded implications. In A. Nerode et al., editors, *Proc. 1st Intl. Workshop on Logic Programming and Non-Monotonic Reasoning*, pp. 211–227. MIT Press, 1991.

[18] P. Mello and A. Natali. Logic Programming in a Software Engineering Perspective. In *Proc. North American Conference on Logic Programming.* MIT Press, 1989.

[19] D. Miller. A Logical Analysis of Modules in Logic Programming. *Journal of Logic Programming*, 79–108, 1989.

[20] D. Miller. Lexical Scoping as Universal Quantification. In *Proc. 6th Intl. Conf. on Logic Programming, Lisboa, 1989.* MIT Press, 1989.

[21] L. Monteiro and A. Porto. Contextual Logic Programming. In *Proc. 6th Intl. Conf. on Logic Programming, Lisboa, 1989.* MIT Press, 1989.

[22] A. Mycroft and R. A. O'Keefe. A Polymorphic Type System for Prolog. *Artificial Intelligence*, 23, 295–307, 1984.

[23] G. Nadathur and D. Miller. An Overview of λ-Prolog. In *Proc. 5th Intl. Conf. on Logic Programming*, pp. 810–827. MIT Press, 1988.

[24] T. C. Przymusinski. On the declarative semantics of deductive databases and logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pp. 193-217. Morgan Kaufmann, 1988.

[25] D. Sanella and L. Wallen. A Calculus for the Construction of Modular Prolog Programs. *Journal of Logic Programming*, 147–177, 1992.

[26] J. D. Ullman. Principles of Database and Knowledge-Base Systems. Vol. II. Computer Science Press, 1989.

From Context-Free to Definite-Clause Grammars (Generalization from Examples)

Juergen Haas
Department of Genetics
University of Pennsylvania
Philadelphia, PA 19104-6145
haas@cbil.humgen.upenn.edu

Bharat Jayaraman
Department of Computer Science
State University of New York at Buffalo
226 Bell Hall
Buffalo, NY 14260
bharat@cs.buffalo.edu

Abstract

This paper discusses the mechanical transformation of an unambiguous context-free grammar (CFG) into a definite-clause grammar (DCG) using a finite set of examples, each of which is a pair $\langle s,\ m\rangle$, where s is a sentence belonging to the language defined by the CFG and m is a semantic representation (meaning) of s. The resulting DCG would be such that it could be executed to compute the semantics for every sentence of the original DCG. Our proposed approach is based upon two key assumptions: (a) the semantic representation language is the ***simply-typed λ-calculus***; and (b) the semantic representation of a sentence is a function (expressed in the typed λ-calculus) of the semantic representations of its parts (***compositionality***). With these assumptions we show that a higher-order DCG can be systematically constructed using a unification procedure for typed λ-terms. The needed procedure differs from the one given by Huet in that the types for variables are not completely known in advance; and it differs from the one used in λProlog in that there is an additional source of nondeterminism in enumerating ***projection*** substitutions. We believe that such a system would simplify the task of building DCGs when the semantic representation involved quantified terms, and could be a useful tool for generating natural query language front-ends for various applications.

1 Motivation

The goal of this work is to develop a system that will take as input an unambiguous ***context-free grammar*** (CFG) and a finite set of pairs $\langle s,\ m\rangle$, where s is a sentence belonging to the language defined by the CFG and m is the semantic representation (meaning) of s, and will produce as output a ***definite clause grammar*** (DCG) (Pereira and Warren 1980, Pereira and Shieber 1987) capable of computing the semantic representations for all sentences of the CFG. We envisage that the system would actually work interactively, by querying the user for the semantic representations

for a series of key sentences (which it determines according to some scheme) and reporting back to the user the synthesized DCG after each sentence until the user accepts the DCG. In order to narrow the search space of possible solutions, we adopt the following two constraints: (1) the semantic representation language is the ***simply typed λ-calculus***; (2) the semantic representation of a sentence is some function (expressed in the typed λ-calculus) of the semantic representations of the phrases that constitute the sentence (***compositionality***). Under these assumptions we believe that, if there is a DCG satisfying the input pairs, it is possible to systematically search for it; if there is no solution, the search may sometimes be nonterminating.

The motivation for our work stems from the fact that it is not easy to manually modify a CFG to obtain a DCG especially when the semantic representations involved quantified terms (as in natural languages). However, by the compositionality principle, the semantic representation of a sentence can be systematically obtained from those of its constituent phrases. Hence, it seems feasible, in principle, to have the computer assist a human in the transition from a CFG to a DCG. A potential use of our proposed system is that it might facilitate rapid prototyping of natural-language interfaces to databases, since the interface could be obtained by defining the syntax along with typical input sentences and their semantic representations. Our proposed use of the simply-typed λ-calculus not only has precedent for natural language semantics (Dowty et al 81, Miller and Nadathur 86), the availability of a unification procedure for simply-typed terms (Huet 75) allows us to reduce the problem of generalization from examples to a unification problem. However, as we shall see later, certain important changes to Huet's procedure are needed in our context, since the types for variables are not completely known in advance.

The remainder of this paper is structured as follows: section 2 outlines the synthesis procedure; section 3 briefly discusses aspects of the synthesis procedure, especially compositionality, termination, multiple solutions, and types; section 4 illustrates the procedure with an example; and section 5 presents the current status and prospects of this work and brief comments on closely related work. Familiarity with the typed λ-calculus and Huet's unification procedure is assumed.

2 Synthesis of DCGs from CFGs and Examples

In the pseudo-code below, we assume, for simplicity of presentation, that a CFG rule has either a single terminal on its rhs or a sequence of one or more nonterminals (in practice, we permit both terminals and nonterminals on the rhs). As in Prolog DCGs, nonterminals are identifiers beginning with a lowercase letter, and terminals are such identifiers surrounded by [and]. As in λProlog (Nadathur and Miller 88), `(F X)` stands for function application and `X\E` stands for λ`X.E` (λ-abstraction). We assume that application is left-associative, i.e., `(F X Y)` is short-hand for `((F X) Y)`. The basic scheme is given below, the top-level procedure being **SYNTH**.

Procedure SYNTH(G)

The procedure **SYNTH** takes as input a CFG and constructs a higher-order DCG after obtaining the semantic representations for sample sentences interactively.

1. Let **G** be an unambiguous CFG having n rules, with start symbol s.

2. Construct the higher-order DCG as follows:

a. If the i-th CFG rule is a_i --> $b_{i1} \ldots b_{ik_i}$, the i-th DCG rule will be

$$(\forall V_1 \ldots V_{k_i})\ a_i((F_i\ V_1 \ldots V_{k_i}), \alpha_{k_i+1}) \text{ --> } b_{i1}(V_1, \alpha_{i1}),\ \ldots\ b_{ik_i}(V_{k_i}, \alpha_{ik_i}),$$

b. If the i-th CFG rule is a_i --> $[t]$, the i-th DCG rule will be

$$a_i(F_i, \alpha_{i1}) \text{ --> } [t].$$

For the sake of clarity, we maintain the types for the function variables $F_1, \ldots, F_n$ explicitly: In 2a, the type of V_i is α_i and the type of F_i is $\alpha_{i1} \rightarrow \ldots \rightarrow \alpha_{ik_i} \rightarrow \alpha_{i(k_i+1)}$. It is important to note that the function variables $F_1, \ldots, F_n$, as well as the type variables α_{ij}, $i = 1, \ldots, n$, $j = 1, \ldots, k_i$ are ***free*** variables of the DCG, i.e., they are not universally quantified like the variables V_i.

3. Solve for the variables F_i in the above DCG as follows.

$E \leftarrow \phi$; *done* $\leftarrow$ *false*; $i \leftarrow 1$;

WHILE *not done* **DO**

a. Generate a set of new sentences se_{ij}, $1 \leq j \leq k_i$, for some finite k_i (selection strategy for these sentences is omitted here). For each se_{ij}, input from the user its semantic representation n_{ij}, a simply-typed term of type t_{ij}.

b. Execute the goal $s(M, t_{ij}, se_{ij}, [\,])$, $1 \leq j \leq k_i$, using the constructed DCG of step 2. For each se_{ij}, let m_{ij}, $1 \leq j \leq k_i$, be the computed answer for variable M.

c. $E \leftarrow E \cup \{m_{ij} = n_{ij}\ :\ 1 \leq j \leq k_i\}$.

d. Call **SOLVE(**E**)**, whose definition is given below. If successful, **SOLVE** nondeterministically returns one of the multiple maximally general unifiers which are possible. Assign *done* $\leftarrow$ *true* if either unification fails, *or* unification succeeds and all sentences of the CFG have been enumerated, *or* unification succeeds and the user accepts the resulting DCG after replacing all variables F_i in the DCG of step 2 according to one of the unifiers of E and reducing all λ-terms to their normal forms.

e. $i \leftarrow i + 1$

END WHILE

4. If unification failed in step **3d**, print "no solution", else print the DCG found.

Procedure SOLVE(E)

Procedure **SOLVE** tries to solve the set of higher-order equations **E** by attempting to find substitutions for the free function variables occurring in **E**.

1. $E \leftarrow \mathbf{E}$; $F \leftarrow \{F_i\ :\ i = 1 \ldots n\}$; $\sigma \leftarrow \phi$ (the empty substitution)

2. WHILE $E \neq \phi$ **DO**

a. Select equation $e = \langle e_1, e_2 \rangle$ from E, and call **SUBST**(e)—note that e_1 is flexible and e_2 is rigid. If **SUBST** succeeds, it returns a substitution term t for the variable V at the head position of e_1. (Definition is **SUBST** is given below.)

b. $\sigma \leftarrow \sigma\{\langle V, t \rangle\}$ (composition of substitutions); $E \leftarrow E\sigma$. Reduce all terms in E and σ to their normal form.

c. $E \leftarrow$ **DECOMP**(E) (definition of **DECOMP** is discussed below).

END WHILE

3. Return $\sigma \downarrow F$ (the *restriction* of σ to the variables F).

Procedure SUBST(e)

Let $\mathbf{e} = \langle e_1, e_2 \rangle$[1], where

$$e_1 = \lambda u_1 \ldots \lambda u_n.(f\ s_1\ s_2\ \ldots\ s_p),$$
$$e_2 = \lambda v_1 \ldots \lambda v_n.(@\ t_1\ t_2\ \ldots\ t_q),$$

and the (simple) type of @ is completely known, say $\delta_1 \rightarrow \ldots \rightarrow \delta_q \rightarrow \beta$, but the type of f may not be completely known—only the number of arguments of f would in general be known. Procedure **SUBST** nondeterministically selects and returns an *imitation* or a *projection* substitution for the head of e_1, provided that the appropriate type constraints are met.

Imitation substitution: applicable only if @ is a constant
$f \leftarrow \lambda w_1 \ldots \lambda w_p.(@\ (h_1\ w_1\ \ldots\ w_p)\ \ldots\ (h_q\ w_1\ \ldots\ w_p))$, where the type of w_i is γ_i, provided the type of f can be unified with $\gamma_1 \rightarrow \ldots \rightarrow \gamma_p \rightarrow \beta$. Each new function variable h_i is assigned a type $\gamma_1 \rightarrow \ldots \rightarrow \gamma_p \rightarrow \delta_i$, for $i = 1, \ldots, q$.

Projection substitutions:
$f \leftarrow \lambda w_1 \ldots \lambda w_p.(w_i\ (h_1\ w_1\ \ldots\ w_p)\ \ldots\ (h_l\ w_1\ \ldots\ w_p))$, for each $1 \leq i \leq p$, provided the type (γ_i) of w_i can be unified with $\epsilon_1 \rightarrow \ldots \rightarrow \epsilon_l \rightarrow \beta$, and the type of f can be unified with $\gamma_1 \rightarrow \ldots \rightarrow \gamma_p \rightarrow \beta$. Each new function variable h_i is assigned a type $\gamma_1 \rightarrow \ldots \rightarrow \gamma_p \rightarrow \epsilon_i$, for $i = 1, \ldots, l$.

While only one imitation substitution is possible, for projection substitutions, there is nondeterminism in the choice of w_i as well as the choice of number of arguments, l. The latter arises because the type γ_i of w_i may not be completely known.

[1] If e_1 has fewer prefix variables than e_2, we assume e_1 is η-expanded so that they have the same number of prefix variables. If it has more prefix variables than of e_2, then there is no unifying substitution.

Procedure DECOMP(E)

E is a set of equations (or disagreement pairs). This procedure is similar to Huet's SIMPL (Huet 1975), except that the types of function variables are determined as the structure of the terms is recursively traversed. We omit presenting the details of this type propagation in this paper, since the needed procedure is similar to that used in λProlog. Note that the right-hand sides of all equations will be closed terms, with known (simple) types, hence this procedure plays a crucial role in propagating type information.

3 Discussion of the Synthesis Technique

We clarify several facets of the synthesis procedure just described:

1. *Compositionality:* The compositionality principle is expressed in step 2 of procedure **SYNTH** by assuming that, in a CFG rule a --> $b_1 \ldots b_k$, the meaning of the nonterminal a is some function F of the meanings of the nonterminals $b_1 \ldots b_k$, where F is expressible in the typed λ-calculus. When terminal symbols are present along with one or more nonterminals on the rhs of a rule, our methodology assumes that the meaning is independent of these terminal symbols; if the semantics of any such terminal $[t]$ is to be taken into account, it should be replaced by a new nonterminal n, and a new rule n --> $[t]$ added to the CFG.

2. *Types*: One of the crucial issues in this synthesis is the determination of types for the free function variables. The lack of complete knowledge of these types in advance marks an important point of departure from Huet's procedure. While the unification procedure of λProlog must also work with polymorphic types, a crucial difference in our work is that there is an additional source of nondeterminism in procedure **SUBST** in enumerating ***projection*** substitutions. In practice, the needed types tend not to be very complex, and therefore the additional nondeterminism may not be a practical problem. Furthermore, since large DCGs would be synthesized in a modular fashion, the number of unknown variables processed could be kept reasonably small. It seems very reasonable to restrict the user-supplied semantic representations to closed λ-terms, in which case we only need a *matching* procedure, rather than a unification procedure. When it is known that terms are of second-order type, we have the pleasant property that there is a finite matching algorithm (Huet and Lang 78). Recently, even third-order matching was also shown to be decidable (Dowek 92), although this decision procedure cannot be directly used to generate matching substitions.

2. *Termination*: In step 3 of **SYNTH**, we incrementally generate a set of equations, where each equation relates the user's chosen semantic representation for a sentence and the semantic representation that would be derived from the higher-order DCG for this sentence. There are three possible outcomes in solving these equations: failure, success, and nontermination. In case of failure, there is no higher-order DCG satisfying the given semantic representations. In case of successful unification and if the CFG generates a finite language, then successful termination is achieved when all sentences have been enumerated. Since the unification procedure is only recursively enumerable, the search may sometimes proceed indefinitely

when there is no solution. If we restricted attention to matching, our problem would reduce to general higher-order matching (beyond order 3), whose decidability is still unknown.

3. *Multiple Solutions*: Since the unification of typed λ-terms could result in multiple maximally general unifiers (i.e., most general unifiers do not always exist), multiple DCG solutions are possible at any stage. We are currently examining criteria that the sample sentences must satisfy so that a unique solution is produced, in the sense that the DCGs corresponding to all other solutions exhibit the same input/output behavior.

4 Example

We illustrate the synthesis by "stepping through" procedure **SYNTH** for a very simple example. For readability, we indicate the types only selectively in this derivation.

(Step **1.**) Assume the CFG is as follows:

```
s --> pn, iv.
pn --> [shrdlu].
pn --> [eliza].
iv --> [runs].
iv --> [halts].
```

(Step **2.**) The DCG resulting from step 2 would be as follows:

```
s((F1 A B)) --> pn(A),iv(B).
pn(F2) --> [shrdlu].
pn(F3) --> [eliza].
iv(F4) --> [runs].
iv(F5) --> [halts].
```

(Step **3a.**) Using the CFG from step 1, the system generates the following sample sentences: `[shrdlu,runs]`, `[eliza,runs]`, and `[shrdlu, halts]`, for which the user provides the corresponding semantic representations: `(run shrdlu)`, `(run eliza)`, and `(halt shrdlu)`, where $\tau($`run`$)$ $= i \rightarrow o$, $\tau($`halt`$) = i \rightarrow o$, $\tau($`shrdlu`$) = i$, and $\tau($`eliza`$) = i$.
(Step **3b.**) Executing each of these sentences on the enhanced CFG (step 2), the following terms are obtained: `(F1 F2 F4)`, `(F1 F3 F4)`, and `(F1 F2 F5)`.
(Step **3c.**) We obtain the following set of higher-order equations:

```
{(F1 F2 F4) = (run shrdlu),
 (F1 F3 F4) = (run eliza),
 (F1 F2 F5) = (halt shrdlu)}.
```

(Step **3d.**) The above equation-set is passed on to procedure **SOLVE**, which in turn calls **SUBST** to obtain a substitution for `F1`, since `F1` is at the head position of the first equation—assumed to

be the chosen equation. There is only one applicable imitation substitution, `K\L\(run (H1 K L))`, since **F1** has two arguments and **run** has one argument. However, this substitution fails to satisfy the third equation. Hence, a projection substitution must be chosen. Since the type of **F1** is $\alpha_1 \rightarrow \alpha_2 \rightarrow o$, the projection substitution must take two arguments. The simplest projection substitutions in this case would be `K\L\L` or `K\L\K`, both of which would eventually lead to failure. The substitution that eventually succeeds is:

```
F1 <- K\L\ (L (H1 K L))
```

Replacing all occurrences of **F1** in the above equations with its substitution and simplifying those terms using the λ-conversion rules leads to the following set of equations:

```
{(F4 (H1 F2 F4)) = (run shrdlu),
 (F4 (H1 F3 F4)) = (run eliza),
 (F5 (H1 F2 F5)) = (halt shrdlu)}
```

DECOMP has no effect in this case since the heads of all left-hand side terms are flexible. However, as a result of the type constraints that come with the substitution term for **F1**, the type of **F4**, namely, α_2, is unified with $\alpha_3 \rightarrow o$, which in turn instantiates the type of **F1** to $\alpha_1 \rightarrow (\alpha_3 \rightarrow o) \rightarrow o$.

Now **F4** is the head of the first equation and **SUBST** is called to provide a substitution for it. Since the type of **F4** is $\alpha_3 \rightarrow o$, the following imitation substitution is applicable:

```
F4 <- K\ (run (H2 K))
```

The type constraints that come with this substitution term imply that the type of the argument `(H2 K)` of **run** is the same as the type of the corresponding argument of **run** in the right-hand side terms, namely i.

Replacing all occurrences of **F4** by its substitution and reducing all terms to their normal form results in the following set of equations:

```
{(run (H2 (H1 F2 K\(run (H2 K))))) = (run shrdlu),
 (run (H2 (H1 F3 K\(run (H2 K))))) = (run eliza),
 (F5 (H1 F2 F5)) = (halt shrdlu)}
```

Applying **DECOMP** to the above equation set we get:

```
{(H2 (H1 F2 K\(run (H2 K)))) = shrdlu,
 (H2 (H1 F3 K\(run (H2 K)))) = eliza,
 (F5 (H1 F2 F5)) = (halt shrdlu)}
```

Next, **SUBST** may choose projection substitution `K\K` for **H2** which transforms the equations to:

```
{(H1 F2 run) = shrdlu,
 (H1 F3 run) = eliza,
 (F5 (H1 F2 F5)) = (halt shrdlu)}
```

Again, **DECOMP** has no effect and we proceed to the next iteration of **SOLVE**. The next substitution chosen by **SUBST** should be the projection substitution `K\L\K` for `H1`, which would yield:

```
{F2 = shrdlu,
 F3 = eliza,
 (F5 F2) = (halt shrdlu)}
```

This implies that both `F2` and `F3` are of type i, which implies `H1` is of type $i \rightarrow (i \rightarrow o) \rightarrow i$. This in turn instantiates the type of `H2` to $i \rightarrow i$, and the type of `F4` to $i \rightarrow o$. Therefore, `F1` will have type $i \rightarrow (i \rightarrow o) \rightarrow o$. The obvious choice for `F2` and `F3` now is `shrdlu` and `eliza`, respectively, which leaves only one equation:

```
{(F5 shrdlu) = (halt shrdlu)}
```

The type of `F5` is easily inferred to be $i \rightarrow o$. `F5` will be replaced by `K\(halt (H3 K))`:

```
{(H3 shrdlu) = (shrdlu)}
```

The projection substitution `K\K` for `H3` completes the derivation. The final substitutions with their types are:

F1: $(i \rightarrow (i \rightarrow o) \rightarrow o)$ = `K\L\(L K)`
F4: $(i \rightarrow o)$ = `run`
H2: $(i \rightarrow i)$ = `K\K`
H1: $(i \rightarrow (i \rightarrow o) \rightarrow i)$ = `K\L\K`
F2: i = `shrdlu`
F3: i = `eliza`
F5: $(i \rightarrow o)$ = `halt`
H3: $(i \rightarrow o)$ = `K\K`

(Step **4**.) Substituting these in the grammar from step 2 yields the following higher-order DCG:

```
s((K\L\(L K) A B)) --> pn(A),iv(B).
pn(shrdlu) --> [shrdlu].
pn(eliza) --> [eliza].
iv(run) --> [runs].
iv(halt) --> [halts].
```

5 Status and Further Work

An implementation of our synthesis procedure has been completed. Using this implementation, we have successfully synthesized larger DCGs than the one shown in this paper, and we are examining the synthesis of a DCG for the natural query language of Chat-80 (Warren and Pereira 1982). There

are several interesting theoretical and practical issues that we have not addressed here: enumeration order for sample sentences and their effect on the synthesis; methodology for writing grammars and semantic representations so that solutions can be efficiently found; constraints from different types of grammars and semantic representations; efficient implementation of higher-order matching and search control; and partial execution of the higher-order DCG, to convert it into a first-order DCG for more efficient execution—we have explored this topic to some extent in (Haas 93).

The techniques needed to develop our proposed system can also lead to a new approach to machine learning as well as program synthesis from examples, by combining higher-order unification with learning from examples. The recent work of Hagiya (Hagiya 90, Hagiya 91) represents an interesting step in this direction. Finally, we note that it appears possible to augment the input CFG with parameters that specify context sensitive features such as number and gender agreement, without affecting the scheme described in this paper. The restriction to unambiguous CFGs is also not an absolute requirement, and it appears possible to extend our approach to ambiguous grammars, which is crucial for general natural language applications.

Acknowledgments

We thank Dale Miller, Fernando Pereira, and the anonymous referees for their helpful comments and suggestions. This research was supported in part by NSF grant CCR 9004357.

References

Dowty, David R., Wall, Robert E. and Peters, Stanley (1981) "Introduction to Montague Semantics," Dordrecht, Holland; Boston: D. Reidel Pub. Co.; Hingham, MA.

Dowek, G. (1992) "Third Order Matching is Decidable," *Seventh Annual LICS*, pp. 2-10, Santa Cruz, CA, June 1992.

Haas, J. (1993) "Automatic Generalization of Semantics," Ph.D. dissertation, Department of Computer Science, SUNY-Buffalo, expected 1993.

Haas, J. and Jayaraman, B. (1992) "Interactive Synthesis of Definite Clause Grammars," *Proc. Jt. Intl. Conf. and Symp. on Logic Programming*, pp. 541-55, MIT Press.

Hagiya, M. (1990) "Programming by Example and Proving by Example using Higher-order Unification," In M. E. Stickel, editor, *Proceedings of the 10th International Conference on Automated Deduction*, pages 588–602, Kaiserslautern, Germany, July 1990. Springer-Verlag LNAI 449.

Hagiya, M. (1991) "From Programming by Example to Proving by Example," In T. Ito and A. R. Meyer, editors, *Proceedings of the International Conference on Theoretical Aspects of Computer Software*, pages 387–419, Sendai, Japan, September 1991. Springer-Verlag LNCS 526.

Huet, G. P. (1975) "A Unification Algorithm for Typed λ-Calculus," *Theoretical Computer Science*,

1 27-57.

Miller, D. and Nadathur, G. (1986) "Some Uses of Higher-Order Logic in Computational Linguistics," *Proc. 24th Annual Meeting of the Assoc. for Computational Linguistics*, pp. 247-255.

Nadathur, G. and Miller, D. (1988) "An Overview of λProlog," *Proc. 5th ICLP*, pp. 810-827.

Nadathur, G. and Miller, D. (1990) "Higher-order Horn clauses," *Journal of the ACM*, pp. 777 – 814.

Pereira, F. C. N., and Warren, D. H. D. (1980) "Definite Clause Grammars for Language Analysis - A Survey of the Formalism and a Comparison with Transition Networks," *Artificial Intelligence*, vol. 13, pp. 231-278.

Pereira, F. C. N. and Shieber, S.M. (1987) "Prolog and Natural-Language Analysis," CSLI.

Warren, D.H.D., and Pereira, F.C.N. (1982) "An efficient easily adaptable system for interpreting natural language queries," *American Journal of Computational Linguistics*, 8 (3-4), 110-22.

Generalization at Higher Types

Robert W. Hasker [1] and Uday S. Reddy
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801 USA
{hasker, reddy}@cs.uiuc.edu

1 Abstract

Finding the most specific generalization of first-order terms is well understood, but the generalization of higher-order terms remains unsolved. We provide a framework for the second-order case by using a categorical structure for terms and identifying a class of useful generalizations called *relevant* generalizations. We show that complete sets of maximally specific generalizations exist and are computable. Such generalizations have an important application for proof modification in automated proof systems.

2 Introduction

Automated proof development systems, including program verification systems, program construction systems, and program transformation systems [4, 10, 2, 15] face the problem of how to incorporate *modifications.* Having constructed a proof for a theorem (or, a program for a specification) as a combination of manual and automated effort, we would certainly not wish to redo the entire effort when the theorem is slightly modified. There is no great damage in redoing the automated part of the proof, but redoing the manual part of the proof manually could be too cumbersome. An ideal automated system should be able to compare the old and new theorems, keep track of the differences, and apply the steps of the old proof to the new theorem as long as they are applicable. We call such a system a *replay* system.

A fundamental problem in building replay systems is drawing analogies between the old and new theorems. The problem can be restated in terms of anti-unification [14, 16]. Given two terms t and u, find the most specific generalization g of the two terms together with the attendant substitutions $\theta : g \to t$ and $\sigma : g \to u$. The triple $\langle g, \theta, \sigma \rangle$, called the *anti-unifier* of t and u, contains the information necessary to relate the subterms of t and u.

If t and u are first-order terms, their first-order anti-unifier can be computed using well-known techniques [14, 16, 8]. However, in modern proof systems the formulas and terms involved are higher-order [7, 11, 10, 4]. Secondly, even if the terms are first-order, the first-order anti-unifier does not contain enough information to relate all corresponding subterms. For instance, if a formula $\mathtt{A}$ is replaced by a conjunction $\mathtt{A} \wedge \mathtt{B}$, the first-order anti-unifier gives the trivial generalization x, loosing the information that $\mathtt{A}$ appears in both the formulas. Another common modification often made is to add parameters to functions and predicates. However, the first-order anti-unifier of $f(t)$

[1] The work by Hasker was supported in part by a grant from Motorola Corp.

and $g(t, u)$ is again trivial. Thus, higher-order generalization is necessary to compute analogies in a replay system.

Surprisingly, even though the first-order anti-unification algorithms has been known since [14, 16], its higher-order counterpart does not seem to have received attention. Recently, [12] gave an algorithm for anti-unifiers for a special class of terms called *patterns* (terms restricted so that only abstraction variables can appear as arguments of a free variable), but the general case is yet unsolved. The pattern restriction precludes using such anti-unifiers in replay systems because it generates nearly the same anti-unifier as in the first-order case. In fact, the difficulties in generalizing higher-order terms while allowing for common subterms are considerable. While first-order terms form a complete lattice with unifiers as infs and anti-unifiers as sups, higher-order terms do not even posses infs. Huet's [8] "algorithm" computes a complete set of minimal unifiers, but the set can be infinite. For the opposite direction of upper bounds, we show that complete sets do not exist, in general. In fact, we believe that the naïve notion of "more general than" used in the first-order case is not meaningful for higher-order terms.

In this paper, we consider the problem of generalization restricted to second-order terms. We define the notion of generality using a categorical framework with substitutions as morphisms between terms. Complete sets of generalizations do not exist even in this setting, but we note that this is due to certain trivial generalizations. By restricting attention to nontrivial generalizations (called *relevant* generalizations), we find that complete sets exist and have interesting properties. We also show that the complete sets are computable and give a semi-practical algorithm for computing them.

3 Notation

We will generalize simply-typed λ-terms [3]. If $C = \uplus_\tau C_\tau$ is the set of constants and $V = \uplus_\tau V_\tau$ the variables, then a term is well-typed if it is consistent with the rules

$$\frac{c \in C_\tau}{c : \tau} \qquad \frac{x \in V_\tau}{x : \tau}$$

$$\frac{t : \tau \to \tau' \quad u : \tau}{t\ u : \tau'} \qquad \frac{x : \tau \quad t : \tau'}{\lambda x.t : \tau \to \tau'}$$

We use the convention that constants are set in `type` and variables in *italics*. We assume all terms are well-typed.

The order of a type τ is defined as

$$\begin{aligned}\text{order}(D_i) &= 1\\ \text{order}(\tau \to \tau') &= \max(1 + \text{order}(\tau), \text{order}(\tau'))\end{aligned}$$

The order of a term is just the order of its type. In this paper, we assume all terms are first or second-order, so all constants are in $C_{D_1 \to \ldots \to D_n}$ and all variables in $V_{D_1 \to \ldots \to D_m}$ for $n, m \geq 1$.

We assume the usual α, β, and η conversion rules. All equivalences between λ-terms and substitutions over λ-terms are assumed to be modulo λ-conversion. Application associates to the

left and $\to$ to the right; parentheses are often dropped when they are not needed. By the Church-Rosser and strong normalization properties of typed λ-calculus (see, *e.g*, [5]), every term of type $D_1 \to \ldots \to D_n \to D_{n+1}$ can be written in the form[2]

$$\lambda x_1.\lambda x_2.\cdots \lambda x_n.hu_1u_2\ldots u_m$$

where each x_i is distinct, $h \in C_\tau \cup V_\tau$, and each u_i is in normal form. We call h the *head* and $\overline{u}$ the *arguments*. Following [8], we say that a term is *flexible* if its head is a free variable and *rigid* otherwise (*i.e*, if it is a constant or a bound variable). We will abbreviate terms in normal form as $\lambda x_1 \cdots x_n.h(u_1,\ldots,u_m)$ or even as $\lambda\overline{x_n}.h(\overline{u_m})$ where $\overline{x_n}$ denotes the sequence $x_1,\ldots,x_n$. If n is 0, then $\overline{x_n}$ is the empty sequence, and if n is arbitrary (but finite) we denote the sequence as just $\overline{x}$. The identity function $\lambda x.x$ is abbreviated as π and the general projection function $\lambda\overline{x_n}.x_k$ as π^n_k. The set of free variables in the term t is $\mathcal{FV}(t)$, and the set of bound variables is $\mathcal{BV}(t)$. The context of u in t is denoted $t[u]$ or alternatively as $t[\alpha - u]$ if its position is relevant.

A substitution θ is a finite map from variables to type-equivalent terms with the domain denoted as $dom(\theta)$ and free variables in the range as $ran(\theta)$. θ_{id} denotes the identity substitution. Application of θ onto term t is variously denoted by $\theta(t)$ and $\theta : t \to u$ (where $u = \theta(t)$). The composition of two substitutions is defined as $\theta \circ \sigma = \lambda t.\theta(\sigma(t))$. To make substitutions easier to read, we will often leave the variables being bound implicit: if the substitution θ is being applied to term u, we may write it as $[\theta(x_1),\ldots,\theta(x_n)]$ where $\langle x_1,\ldots,x_n\rangle$ are the free variables listed in the order they occur when reading u from left to right.

4 The category of generalizations

First-order generalizations can be compared using the preorder $v \leq u \iff \exists\theta.\theta : v \to u$. This ordering is adequate because the substitution is unique, but in the higher-order case it often is not. Category theory provides a framework which supports distinguishing between substitutions.

In this section we examine the category of terms show that it is inadequate. This leads to the category of generalizations and a discussion of its inadequacies.

Definition 4.1 Given a type τ, the category $\mathbf{T}_\tau$ has as objects terms of type τ. The arrows of $\mathbf{T}_\tau$ are given by substitutions $\theta : t \to u$ such that $dom(\theta) = \mathcal{FV}(t)$, $ran(\theta) = \mathcal{FV}(u)$, and $\theta(t) = u$. The composition is substitution composition and the identity arrows are identity substitutions.

We often leave the type subscript τ implicit. When $\theta : t \to u$ we say that t is *more general* than u (or conversely, u is *more specific* than t). But, θ indicates in what way t is more general than u. For first-order terms, $\mathbf{T}$ is a preorder; *i.e*, there is at most one substitution between any two terms. For second-order terms, this is not the case; for example,

$$[\lambda x.g(x,\mathsf{a})] \text{ and } [\lambda x.g(x,x)] : f\mathsf{a} \to g(\mathsf{a},\mathsf{a})$$

That is, for the second order case, a term may generalize another in multiple ways. This is the motivation for considering categories instead of preorders.

[2] Note that such forms are in normal form with respect to β, but not η.

Definition 4.2 If $a \in \mathbf{T}$ is a term, the category $\mathbf{G}(a)$—of *generalizations* of a—has as its objects substitutions $\theta : t \rightarrow a$ for $t \in \mathbf{T}$. A (generalization) morphism $\rho : (\theta_1 : t_1 \rightarrow a) \rightarrow (\theta_2 : t_2 \rightarrow a)$ is a substitution $\rho : t_1 \rightarrow t_2$ such that the following triangle commutes:

$$\begin{array}{ccc} t_2 & \xrightarrow{\theta_2} & a \\ \uparrow{\scriptstyle \rho} & \nearrow{\scriptstyle \theta_1} & \\ t_1 & & \end{array}$$

($\mathbf{G}(a)$ is often called the "slice category" $\mathbf{T}_\tau/a$.)[3]

This definition can be extended to generalizations of multiple terms. We show the binary case:

Definition 4.3 If $a_1, a_2 \in \mathbf{T}$, the category $\mathbf{G}(a_1, a_2)$ has as its objects pairs of substitutions $\langle \theta_1 : t \rightarrow a_1, \theta_2 : t \rightarrow a_2 \rangle$. A morphism

$$\rho : \langle \theta_1 : t \rightarrow a_1, \theta_2 : t \rightarrow a_2 \rangle \rightarrow \langle \sigma_1 : u \rightarrow a_1, \sigma_2 : u \rightarrow a_2 \rangle$$

is a substitution $\rho : t \rightarrow u$ that is a generalization morphism in both $\mathbf{G}(a_1)$ and $\mathbf{G}(a_2)$. That is, the following diagram commutes:

$$\begin{array}{ccccc} a_1 & \xleftarrow{\sigma_1} & u & \xrightarrow{\sigma_2} & a_2 \\ & \nwarrow{\scriptstyle \theta_1} & \uparrow{\scriptstyle \rho} & \nearrow{\scriptstyle \theta_2} & \\ & & t & & \end{array}$$

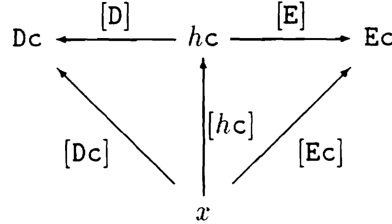
As an aside, note that $\mathbf{G}(a_1, a_2)$ is the pullback $\mathbf{G}(a_1) \times_{\mathbf{T}} \mathbf{G}(a_2)$ in **Cat**. That is, if *src* is the forgetful functor, the diagram

$$\begin{array}{ccc} \mathbf{G}(a_1, a_2) & \xrightarrow{\pi_2} & \mathbf{G}(a_2) \\ {\scriptstyle \pi_1}\downarrow & & \downarrow{\scriptstyle src} \\ \mathbf{G}(a_1) & \xrightarrow{src} & \mathbf{T}_\tau \end{array}$$

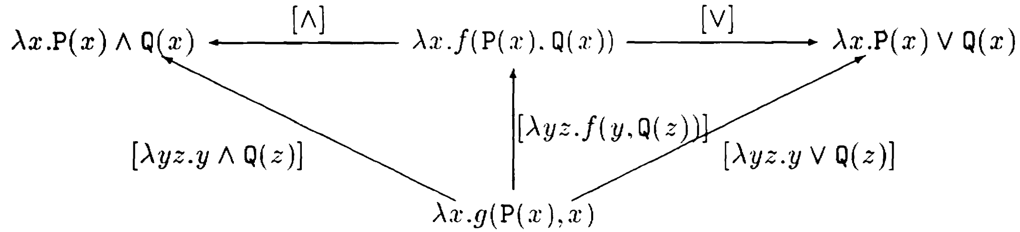
commutes.

[3]Note that $\mathbf{T}_\tau$ can itself be treated as a slice category $\mathbf{Type}/\tau$ where **Type** is the category of types [9, 6].

Examples 4.4 The following examples illustrate that generalizations which include common subterms are more specific:



These generalizations are not isomorphic because the only substitution from hc to x, $\{h \mapsto \lambda y.x\}$, is not a generalization morphism. In comparison, [12] disallows the more specific of the two generalizations because hc is not a valid pattern. The only generalization meeting the pattern restriction is $\langle[\mathtt{Dc}] : x \mapsto \mathtt{Dc},\ [\mathtt{Ec}] : x \mapsto \mathtt{Ec}\rangle$, thus patterns do not capture common subterms.



These are not isomorphic because there is no substitution from $\lambda x.f(\mathtt{P}(x), \mathtt{Q}(x))$ to $\lambda x.g(\mathtt{P}(x), x)$.

Examples 4.5 It is also instructive to examine generalizations which are unrelated by morphisms. The first illustrates that for two generalizations to be related, subterms must be used consistently:

$$[\lambda x.\mathtt{D}(x, \mathtt{b})] : f\mathtt{a} \rightarrow \mathtt{D}(\mathtt{a}, \mathtt{b}) \quad \text{and} \quad [\lambda x.\mathtt{D}(\mathtt{a}, x)] : g\mathtt{b} \rightarrow \mathtt{D}(\mathtt{a}, \mathtt{b})$$

These are unrelated because any generalization morphism would have to eliminate the a (from fa) or b (from gb). The second example illustrates that different substitutions give rise to unique generalizations:

$$[\lambda xy.\mathtt{E}(x, x, y)] : \lambda z.f(z, z) \rightarrow \lambda z.\mathtt{E}(z, z, z)$$

$$[\lambda xy.\mathtt{E}(y, x, x)] : \lambda z.g(z, z) \rightarrow \lambda z.\mathtt{E}(z, z, z)$$

These are unrelated because the substitutions project distinct arguments.

Two generalizations g_1 and g_2 are *isomorphic*, written $g_1 \cong g_2$, if there are $\rho : g_1 \rightarrow g_2$ and $\rho_{op} : g_2 \rightarrow g_1$ such that $\rho \circ \rho_{op} = \theta_{id} = \rho_{op} \circ \rho$. We can show that isomorphisms are renamings.

Definition 4.6 ([12]) θ is a *renaming* iff for all $f \in dom(\theta)$, $\theta(f) = \lambda\overline{x}.h(\overline{x'})$ where h is a variable and $\overline{x'}$ is a permutation of $\overline{x}$.

Lemma 4.7 $g_1 \cong g_2$ by $\rho : g_1 \rightarrow g_2$ and $\rho_{op} : g_2 \rightarrow g_1$ iff ρ and ρ_{op} are renamings.

This follows from the observation that whenever $\theta(\sigma(f)) = f$ and $\sigma(f) = \lambda\overline{x}.t$, t must be flexible and all $x_i \in \overline{x}$ occur in t.

Observation 4.8 $([a] : f \to a)$ is initial in $\mathbf{G}(a)$.

This is because there is only one substitution between $\lambda\overline{x}.f(\overline{x})$ and any term. Since θ_{id} is a left identity,

Observation 4.9 $\theta_{id} : a \to a$ is terminal in $\mathbf{G}(a)$.

However, the morphisms of $\mathbf{G}(a)$ are not always unique:

Example 4.10

$$
\begin{array}{ccc}
\lambda x.h(x) & \xrightarrow{[\pi]} & \lambda x.x \\
{\scriptstyle [h,\pi],\text{ or } [\pi,h]}\uparrow & \nearrow{\scriptstyle [\pi,\pi]} & \\
\lambda x.f(g(x)) & &
\end{array}
$$

Another difficulty is that $\mathbf{G}$ is not well-behaved with respect to maximal objects. Ideally, the maximally specific generalizations of any two terms a and b would be the maximal objects of $\mathbf{G}(a, b)$. However, the maximal objects are often undefined. The following examples show that the sources of maximal objects have unbounded depth and width. We also show that the arbitrarily large terms are not isomorphic to smaller terms, thus defining maximal objects up to an isomorphism would not be sufficient.

Example 4.11 Consider $\mathbf{G}(\mathsf{D}a, \mathsf{E}b)$, where a and b are arbitrary terms:

$$
\begin{array}{ccccc}
\mathsf{D}a & \xleftarrow{[\mathsf{D},\pi_1^2]} & f(g(a,b)) & \xrightarrow{[\pi_1^1,\mathsf{E}\circ\pi_2^2]} & \mathsf{E}b \\
 & {\scriptstyle [\mathsf{D}\circ\pi_1^2]}\nwarrow & \uparrow{\scriptstyle [f\circ g]} & \nearrow{\scriptstyle [\mathsf{E}\circ\pi_2^2]} & \\
 & & h(a,b) & &
\end{array}
$$

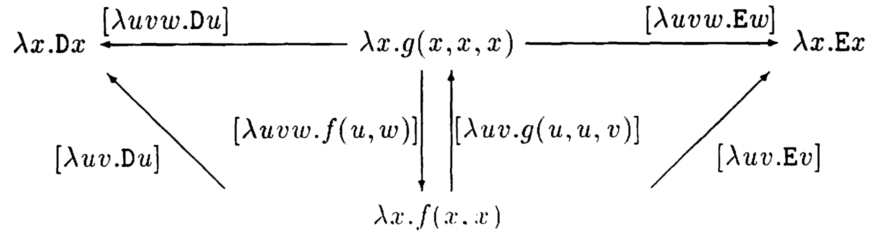
Note that the two generalizations are not isomorphic because there is no generalization morphism in the opposite direction. If ρ was such a morphism, then $head(\rho(f)) = h$ or $\rho(f) = \pi$, but neither choice allows both sides to commute simultaneously. Similarly, g can be mapped to $f' \circ g'$ and so on. A generalization morphism in the opposite direction can be found after a few repetitions of the pattern, but the generalizations remain nonisomorphic.

Example 4.12 $\mathbf{G}(\mathsf{c}, \mathsf{d})$ contains

$$
\begin{array}{ccccc}
[\mathsf{c}] & \xleftarrow{[\pi_1^2]} & g(\mathsf{c},\mathsf{d},\mathsf{e}) & \xrightarrow{[\pi_2^2]} & \mathsf{d} \\
 & {\scriptstyle [\pi_1^2]}\nwarrow & {\scriptstyle [\lambda xyz.f(x,y)]}\downarrow\;\uparrow{\scriptstyle [\lambda xy.g(x,y,\mathsf{e})]} & \nearrow{\scriptstyle [\pi_2^2]} & \\
 & & f(\mathsf{c},\mathsf{d}) & &
\end{array}
$$

Again, these are not isomorphic. This example can also be generalized to an arbitrary number of subterms in place of e. A similar situation occurs when bound variables are repeated arbitrarily often:

Example 4.13 $\mathbf{G}(\lambda x.\mathtt{D}x, \lambda x.\mathtt{E}x)$ contains



5 Relevant generalizations

These examples show that while **G** may be more a suitable category in which to find maximal generalizations than **T**, it is not ideal. We can improve on **G** by restricting attention to only those generalizations which are *relevant*, where relevance means that each subterm is useful in forming the generalization. In particular, the following definitions permit variables only when they are necessary and permit rigid subterms only when they are actually used.

Example 4.11 suggests disallowing nested flexible subterms. We use the following definitions:

Definition 5.1 A generalization $\theta : t \to a$ is said to be *redundant* if t has a subterm of the form $f(\ldots, g(\ldots), \ldots)$ and $\theta(f) \neq f$ or $\theta(g) \neq g$. We say that a generalization is *condensed* if it is not redundant.

A variable in a condensed generalization must occur either at the outermost position or as an argument of a constant. This bounds the depth of terms.

Examples 4.12 and 4.13 illustrate that we must limit the number of times subterms can appear. The solution is to disallow most substitutions which eliminate subterms.

Definition 5.2 A substitution $\theta : t[f(\overline{u})] \to a$ is said to *eliminate* u_k if $\theta(f) = \lambda\overline{x}.M$ and x_k does not occur in M.

That is, a subterm of $f(\overline{u})$ is eliminated if $\theta(f)$ is independent of the corresponding abstraction. This is a generalization of the definition of elimination introduced in [13].

Definition 5.3 A subterm u_k of $t[H(\overline{u})]$ is *uneliminable* if

i. $u_k \notin \mathcal{BV}(t)$, or

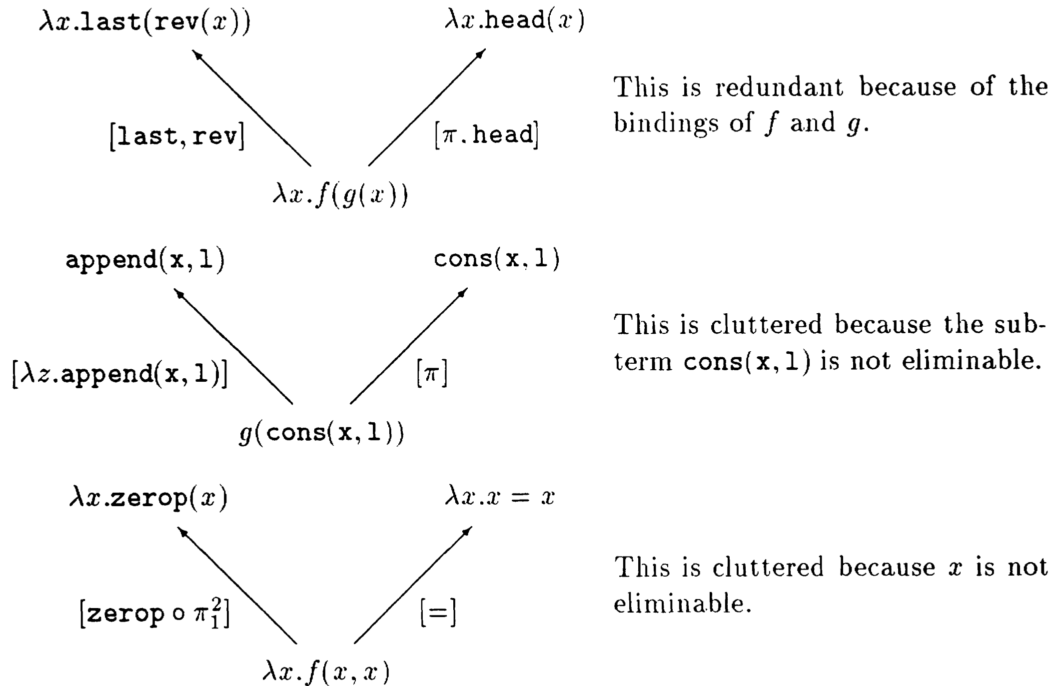
ii. $u_k \in \mathcal{BV}(t)$ and $u_k = u_{k'}$ for $k' \neq k$.

Definition 5.4 A generalization $\theta : t \to a$ in $\mathbf{G}(a)$ is *cluttered* if for some $f \in dom(\theta)$, $\theta(f)$ eliminates an uneliminable subterm.

By disallowing cluttered generalizations, we bound the width of terms. However, some generalizations which eliminate bound variables *are* allowed so that $\lambda xy.x$ and $\lambda xy.y$ can be generalized (using $\lambda xy.f(x, y)$).

Definition 5.5 A generalization $g \in \mathbf{G}(a)$ is said to be *relevant* if it is *condensed* and ***not cluttered.*** Let $\mathbf{R}(a)$ be the full subcategory of $\mathbf{G}(a)$ consisting of relevant generalizations. Similarly, let $\mathbf{R}(a, b)$ be the full subcategory of $\mathbf{G}(a, b)$ consisting of pairs of relevant generalizations.

Examples 5.6 The following generalizations are *irrelevant*:



If we ignore renamings,

Lemma 5.7 $\mathbf{R}(a)$ is finite.

This is because the number of constants is limited by the size of a, which limits the number of free variables (since each must be separated by a constant), and the sum of the two limits the number of bound variables. Since $\mathbf{R}(a, b)$ contains only pairs of objects from both $\mathbf{R}(a)$ and $\mathbf{R}(b)$,

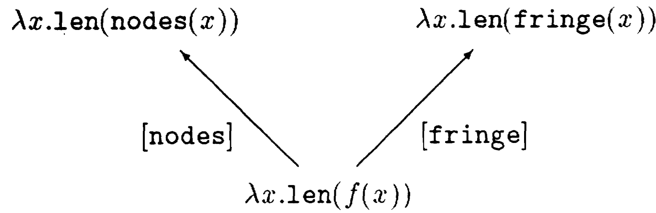
Corollary 5.8 $\mathbf{R}(a, b)$ is finite.

Since the most specific generalization may not be unique, we define the set of maximally specific generalizations:

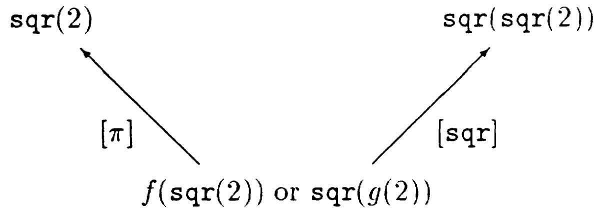
Definition 5.9 $\text{MSG}(a, b)$ is the least set of generalizations in $\mathbf{R}(a, b)$ such that $\forall g \in \mathbf{R}(a, b), \exists g' \in \text{MSG}(a, b)$ such that $g \rightarrow g'$ (up to an isomorphism).

Note that the least set exists because if $g \to g_1'$ and $g \to g_2'$ where $g_1' \leftrightarrow g_2'$, then g_1' and g_2' are isomorphic by Lemma 4.7.

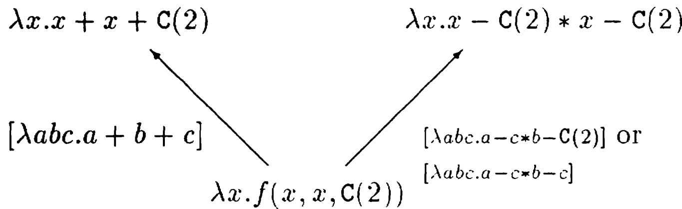
Examples 5.10 Some maximal (relevant) generalizations:



This is maximally specific because len is the only common constant.



This illustrates how multiple maximal objects can arise when there are different possible pairings.



Different ways of instantiating can also lead to multiple generalizations. Note that there are more generalizations as well.

5.1 Properties of R

G is not a preorder because its morphisms are not always unique. In this section, we show that **R** is a preorder. This property is interesting in itself and also helps in showing the correctness of our algorithm to compute the complete set of most specific generalizations. All the results of this section extend to the binary case (and multiple term cases) because the morphisms of $\mathbf{R}(a, b)$ are a subset of the morphisms of $\mathbf{R}(a)$ and $\mathbf{R}(b)$.

Lemma 5.11 Whenever $g_1, g_2 \in \mathbf{R}(a)$ and $\rho : g_1 \to g_2$, $\rho : src(g_1) \to src(g_2)$ is relevant.

Proof Let g_1 be $\theta_1 : t_1 \to a$ and g_2 be $\theta_2 : t_2 \to a$. If $\rho : t_1 \to t_2$ is not relevant, ρ eliminates a subterm w of t_1. But then $\theta_2 \circ \rho$ eliminates w; this contradicts $\theta_2 \circ \rho = \theta_1$. □

Theorem 5.12 $\mathbf{R}(a)$ is a preorder.

We need to show that there is at most one morphism between any two generalizations.[4] Consider the commutative diagram

$$\begin{array}{ccc} u & \xrightarrow{\sigma} & a \\ \rho_1 \uparrow\uparrow \rho_2 & \nearrow \theta & \\ t[f(\overline{w})] & & \end{array}$$

in $\mathbf{R}(a)$ and let $f(\overline{w})$ be the outermost subterm of t such that f is a free variable and $\rho_1(f) \neq \rho_2(f)$. Observe that since $\rho_1 : t \rightarrow u$ and $\rho_2 : t \rightarrow u$ are relevant, there is at least one occurrence of $\rho_1(f(\overline{w}))$ and $\rho_2(f(\overline{w}))$ in u. Also observe that this occurrence must be the same for both ρ_1 and ρ_2 since f is the outermost variable for which ρ_1 and ρ_2 differ. Call this occurrence u'. The key lemma for showing that f does not exist is

Lemma 5.13 If $\rho_1(f) = \pi_i^n$, then $\rho_2(f) = \pi_i^n$.

Proof Since $\rho_1 : t \rightarrow u$ is relevant, each $\overline{w}$ other than w_i must be eliminable, so they are all projections different from each other and different from w_i. There are three cases:

1. *head*(w_i) is a constant: w_i is uneliminable, so $\rho_2(f) = \pi_i^n$.
2. *head*(w_i) is a free variable: this case is impossible because $\rho_1 : t \rightarrow a$ is condensed.
3. $w_i \in \mathcal{BV}(t)$: $u' = w_i$, so since there is no other $w_j = w_i$, $\rho_2(f) = \pi_i^n$.

□

This along with the existence of a $u' = \rho_1(f(\overline{w}))$ gives us

Lemma 5.14 *head*$(\rho_1(f)) =$ *head*$(\rho_2(f))$.

Proof of 5.12 We show $\rho_1(f) = \rho_2(f)$. Suppose $\rho_1(f) = \lambda\overline{x}.K(\overline{M})$ for some constant K. Then $\rho_2(f)$ is $\lambda\overline{x}.K(\overline{N})$ by Lemma 5.14, and we use induction on the depth of substitutions (using a multiset ordering) to show $\overline{M} = \overline{N}$ by constructing the commutative diagram

$$\begin{array}{ccc} u & \xrightarrow{\sigma} & a \\ \rho_1' \uparrow\uparrow \rho_2' & \nearrow \theta' & \\ v & & \theta \\ \uparrow \phi & & \\ t[f(\overline{w})] & & \end{array}$$

[4] Note that renamings are *not* allowed.

in $\mathbf{R}(a)$ such that $\rho_1 = \rho_1' \circ \phi$ and $\rho_2 = \rho_2' \circ \phi$. The following function is used to ensure $\theta' : v \to a$ is relevant:

Definition 5.15

$$\begin{array}{l} \mathit{projected}(\overline{x_n}, t) = \\ \quad \text{the } \langle x_{j_1}, \ldots, x_{j'_n} \rangle \text{ such that each } x_{j_k} \in \{\overline{x_n}\}, \text{ all } j_k < j_{k+1}, \text{ and } x_{j_k} \text{ occurs in } t \end{array}$$

Then let

$$\begin{array}{rcl} \theta(f) & = & \lambda\overline{x_n}.K(\overline{P}) \\ \nu_i & = & \mathit{projected}(\overline{x_n}, M_i) \cup \mathit{projected}(\overline{x_n}, N_i) \\ \phi & = & \{f \mapsto \lambda\overline{x_n}.K(h_1(\nu_1), \ldots, h_m(\nu_m))\} \\ \rho_1' & = & \rho_1 \setminus f \cup \{h_1 \mapsto \lambda\nu_1.M_1, \ldots, h_m \mapsto \lambda\nu_m.M_m\} \\ \rho_2' & = & \rho_2 \setminus f \cup \{h_1 \mapsto \lambda\nu_1.N_1, \ldots, h_m \mapsto \lambda\nu_m.N_m\} \\ \theta' & = & \theta \setminus f \cup \{h_1 \mapsto \lambda\nu_1.P_1, \ldots, h_m \mapsto \lambda\nu_m.P_m\} \end{array}$$

(where each $\overline{h_m}$ is a free variable occurring nowhere else). Note that $\mathit{projected}(\overline{x_n}, P_i)$ must be a subsequence of ν_i since σ cannot introduce abstractions. Furthermore, if N_i eliminates x_j in ν_i, then w_j is eliminable. This is because if x_j does not appear in N_i, it must be eliminated by σ from M_i, so it must be in the scope of a free variable f'' in M_i. Since $\sigma : u \to a$ is uncluttered and w_j is eliminated by $\sigma(f'')$, w_j must be eliminable. Thus $\theta' : v \to a$ is relevant and we can use induction to show $\rho_1'(f) = \rho_2'(f)$.

A similar argument is used when the head of both $\rho_1(f)$ and $\rho_2(f)$ is a free variable, g, except that the details must be modified to ensure $\theta' : v \to a$ is condensed. Observe that the arguments to g must be rigid terms (unless $\rho_1(f) = f$ and $\rho_1(g) = g$, in which case $\rho_2(f) = f$ and $\rho_2(g) = g$ because $\rho_2 : v \to u$ is relevant). Thus $\rho_1(f) = \lambda\overline{x_n}.g(\overline{M_m})$ and $\rho_2(f) = \lambda\overline{x_n}.g(\overline{N_m})$ where $M_i = G_i(\overline{r_{i,p_i}})$ and $N_i = H_i(\overline{s_{i,q_i}})$. We first show that for each k, $H_k = G_k$. Since $\sigma \circ \rho_1 = \theta = \sigma \circ \rho_2$, $\sigma(g \circ \langle \ldots, G_k, \ldots \rangle) = \sigma(g \circ \langle \ldots, H_k, \ldots \rangle)$ and so $\sigma(G_k) = \sigma(H_k)$. Thus $G_k = H_k$ since both are rigid.

We use induction to show that the rest of $\rho_1(f)$ and $\rho_2(f)$ are the same. Assuming $\theta(f)$ is $\lambda\overline{x_n}.g(H_1(r'_{1,1}, \ldots, r'_{1,p_1}), \ldots, H_m(r'_{m,1}, \ldots, r'_{m,p_m}))$, let

$$\begin{array}{rcl} \nu_{i,j} & = & \mathit{projected}(\overline{x_n}, r_{i,j}) \cup \mathit{projected}(\overline{x_n}, s_{i,j}) \\ \phi & = & \{f \mapsto \lambda\overline{x_n}.g(H_1(h_{1,1}(\nu_{1,1}), \ldots, h_{1,p_1}(\nu_{1,p_1})), \ldots, \\ & & \qquad H_m(h_{m,1}(\nu_{m,1}), \ldots, h_{m,p_m}(\nu_{m,p_m})))\} \\ \rho_1' & = & \rho_1 \setminus f \cup \{h_{1,1} \mapsto \lambda\nu_{1,1}.r_{1,1}, \ldots, h_{m,p_m} \mapsto \lambda\nu_{m,p_m}.r_{m,p_m}\} \\ \rho_2' & = & \rho_2 \setminus f \cup \{h_{1,1} \mapsto \lambda\nu_{1,1}.s_{1,1}, \ldots, h_{m,p_m} \mapsto \lambda\nu_{m,p_m}.s_{m,p_m}\} \\ v & = & \phi(t) \\ \theta' & = & \theta \setminus f \cup \{h_{1,1} \mapsto \lambda\nu_{1,1}.r'_{1,p_1}, \ldots, h_{m,p_m} \mapsto \lambda\nu_{m,p_m}.r'_{m,p_m}\} \end{array}$$

Again $\rho_1' = \rho_2'$ by induction, hence $\rho_1 = \rho_2$. □

Since morphisms are unique and θ_{id} is a morphism from any generalization to itself,

Corollary 5.16 $\mathbf{R}(a)$ is a partial order.

This allows us to introduce the following notation:

Definition 5.17 Whenever $g_1 \rightarrow g_2$ is in $\mathbf{R}(a)$, we say g_1 is *less specific* (or, equivalently, *more general*) than g_2. This is written as $g_1 \leq g_2$. Furthermore, we write $g_1 < g_2$ if $g_2 \rightarrow g_1$ is not in $\mathbf{R}(a)$.

6 Computing MSG

$\mathbf{R}(a, b)$ is finite, so since second-order matching is decidable and λ-terms are recursively enumerable, we can compute MSG(a, b) by generating $\mathbf{R}(a, b)$ and comparing all its objects against each other. Thus, MSG(a, b) is computable, albeit inefficiently. A more practical algorithm is suggested by the observation that when the substitutions contain a common subterm, then they can be made more specific by factoring out the common term.

The steps for specializing generalizations of a and b are given by the following rewrite relation $\longrightarrow$. The algorithm is restricted to generalizing ground terms; non-ground terms can be handled by "freezing" the variables; that is, replacing them by unique constants. The $\longrightarrow$ steps maintain the invariants

$$\begin{array}{l} \theta_1 : t \rightarrow a \\ \theta_2 : t \rightarrow b \\ \langle \theta_1 : t \rightarrow a,\ \theta_2 : t \rightarrow b \rangle \text{ is relevant} \\ \text{if } g_1 \longrightarrow g_2,\ g_1 < g_2 \end{array}$$

To compute MSG(a, b), we start with the initial object of $\mathbf{R}(a, b)$ and continue specializing the generalization until no $\longrightarrow$ step is applicable. To simplify the notation, we represent each generalization $\langle \theta_1 : t \rightarrow a,\ \theta_2 : t \rightarrow b \rangle$ by the triple $\langle t, \theta_1, \theta_2 \rangle$.

Delete-variable Variables with the same binding in both substitutions can be removed:

$$\langle t, \theta_1 \cup \{f \mapsto M\}, \theta_2 \cup \{f \mapsto M\} \rangle \quad \longrightarrow \quad \langle \{f \mapsto M\}(t), \theta_1, \theta_2 \rangle$$

Merge Likewise, variables with the same bindings within each substitution can be merged:

$$\begin{array}{lcl} \langle t, & & \langle \{g \mapsto f\}(t), \\ \quad \theta_1 \cup \{f \mapsto M, g \mapsto M\}, & \longrightarrow & \quad \theta_1 \cup \{f \mapsto M\}, \\ \quad \theta_2 \cup \{f \mapsto N, g \mapsto N\} \rangle & & \quad \theta_2 \cup \{f \mapsto N\} \rangle \end{array}$$

Delete-abstraction Subterms which are not projected by either substitution can be eliminated:

$$\begin{array}{lcl} \langle t, & & \langle \{f \mapsto \lambda \overline{x} z \overline{y}. f'(\overline{x}, \overline{y})\}(t), \\ \theta_1 \cup \{f \mapsto \lambda \overline{x} z \overline{y}. M\}, & \longrightarrow & \theta_1 \cup \{f' \mapsto \lambda \overline{x} \overline{y}. M\}, \\ \theta_2 \cup \{f \mapsto \lambda \overline{x} z \overline{y}. N\} \rangle & & \theta_2 \cup \{f' \mapsto \lambda \overline{x} \overline{y}. N\} \rangle \end{array}$$

where z does not occur in either M or N.

Factor-constant Constants that appear in both substitutions can be factored. This step is complicated because it must introduce new function variables for generalizing the subterms and it must not create cluttered terms.

$$\begin{array}{lcl} \langle t, & & \langle \{f \mapsto \lambda\overline{x}.f'(K(h_1(\nu_1),\ldots,h_n(\nu_n)),\nu_0)\}(t), \\ \theta_1 \cup \{f \mapsto \lambda\overline{x}.M[K(\overline{u})]\}, & \longrightarrow & \theta_1 \cup \{f' \mapsto \lambda z\nu_0.M[\forall\alpha \in \hat{\alpha}, \alpha \leftarrow z], \\ & & \quad h_1 \mapsto \lambda\nu_1.u_1,\ \ldots,\ h_n \mapsto \lambda\nu_n.u_n\}, \\ \theta_2 \cup \{f \mapsto \lambda\overline{x}.N[K(\overline{v})]\}\rangle & & \theta_2 \cup \{f' \mapsto \lambda z\nu_0.N[\forall\beta \in \hat{\beta}, \beta \leftarrow z], \\ & & \quad h_1 \mapsto \lambda\nu_1.v_1,\ \ldots,\ h_n \mapsto \lambda\nu_n.v_n\}\rangle \end{array}$$

where

K	is a constant,
n	is the arity of K,
h_i	occurs no where else (for $1 \leq i \leq n$),
$\hat{\alpha}$	is a nonempty subset of the positions at which $K(\overline{u})$ occurs in M,
$\hat{\beta}$	is a nonempty subset of the positions at which $K(\overline{v})$ occurs in N,
ν_0	$= \mathit{projected}(\overline{x}, M[\forall\alpha \in \hat{\alpha}, \alpha \leftarrow z]) \cup \mathit{projected}(\overline{x}, N[\forall\beta \in \hat{\beta}, \beta \leftarrow z])$ (see Definition 5.15 for *projected*), and
ν_k	$= \mathit{projected}(\overline{x}, u_k) \cup \mathit{projected}(\overline{x}, v_k)$.

If the new θ_1 or θ_2 of some h_k (or f') would eliminate an uneliminable subterm, then this step is not applicable (with the chosen $\hat{\alpha}$ and $\hat{\beta}$) because it would form a cluttered generalization.

Factor-abstraction Repeated bound variables can be factored in much the same way as constants except that there is no need to introduce new free variables:

$$\begin{array}{lcl} \langle t, & & \langle \{f \mapsto \lambda\overline{x}.f'(x_i,\overline{x})\}(t), \\ \theta_1 \cup \{f \mapsto \lambda\overline{x}.M[x_i]\}, & \longrightarrow & \theta_1 \cup \{f' \mapsto \lambda z\overline{x}.M[\forall\alpha \in \hat{\alpha}, \alpha \leftarrow z]\}, \\ \theta_2 \cup \{f \mapsto \lambda\overline{x}.N[x_i]\}\rangle & & \theta_2 \cup \{f' \mapsto \lambda z\overline{x}.N[\forall\beta \in \hat{\beta}, \beta \leftarrow z]\}\rangle \end{array}$$

where x_i is in $\overline{x}$, x_i occurs in at least one other position in *both* M and N, and $\hat{\alpha}$ and $\hat{\beta}$ are proper, nonempty subsets of the positions at which x_i occurs. ($\hat{\alpha}$ and $\hat{\beta}$ must be proper subsets so that the new generalization is not cluttered.)

Using $\longrightarrow$, the set MSG can be computed by *gen* defined as

$$\mathit{gen}(a,b) \quad = \quad \{g \mid \langle f, [a], [b]\rangle \longrightarrow^* g, \text{ and } \nexists g'.g \longrightarrow g'\}$$

where $\longrightarrow^*$ is the transitive closure of $\longrightarrow$.

This algorithm is expensive because it requires exponential time and recomputes the same generalizations in different ways. Furthermore, some pairs of terms have an exponential number of generalizations, so there is no polynomial-time algorithm based on the size of the input. It is not yet clear if a polynomial-time algorithm exists based on the number of generalizations.

The proof this algorithm's correctness depends upon showing that the set of $\longrightarrow$ rules completely specifies when one generalization is strictly less instantiated than another. First we give some lemmas:

Lemma 6.1 If $g_1 \in \mathbf{R}(a,b)$ and $g_1 \longrightarrow g_2$, then $g_2 \in \mathbf{R}(a,b)$.

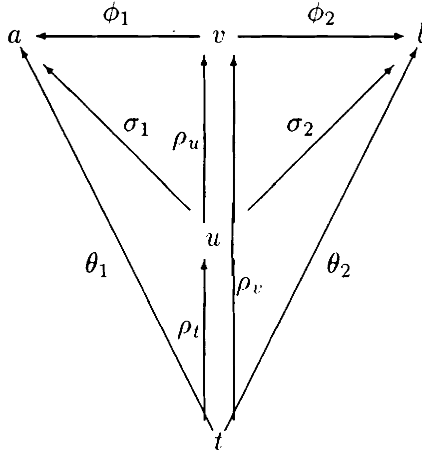
Lemma 6.2 Whenever $g_1, g_2 \in \mathbf{R}(a,b)$ and $g_1 \longrightarrow g_2$, $g_1 < g_2$.
Proof Observe that each step is of the form $\langle t, \theta_1, \theta_2 \rangle \longrightarrow \langle \rho(t), \theta_1', \theta_2' \rangle$ where ρ is a generalization morphism. This shows $g_1 \leq g_2$. Furthermore, ρ is not a renaming substitution, so by Theorem 5.12 there is no $\rho_{op} : g_2 \to g_1$. □

Finally, we show that $\longrightarrow$ steps do not reduce the number of possible generalizations. That is, given a specific generalization, the set of $\longrightarrow$ steps completely covers all maximal generalizations which are more instantiated than the given one.

Lemma 6.3 Whenever $g_v' \in \mathrm{MSG}(a,b)$, $g_t \in \mathbf{R}(a,b)$, and $g_t < g_v'$, there is a $g_u \in \mathbf{R}(a,b)$ and a $g_v \cong g_v'$ such that $g_t \longrightarrow g_u$ and $g_u \leq g_v$.
Proof Assume

$$\begin{aligned} g_t &= \langle \theta_1 : t \to a,\ \theta_2 : t \to b \rangle \\ g_u &= \langle \sigma_1 : u \to a,\ \sigma_2 : u \to b \rangle \\ g_v &= \langle \phi_1 : v \to a,\ \phi_2 : v \to b \rangle \end{aligned}$$

Then the following diagram illustrates this lemma:



Choose an $f \in dom\rho_v$ such that $\rho_v(f)$ is not a renaming substitution unless all substitutions are renamings. Let

$$\begin{aligned} \rho_v(f) &= \lambda\overline{x}.H(\overline{w}) \\ \theta_1(f) &= \lambda\overline{y}.M \\ \theta_2(f) &= \lambda\overline{y}.N \end{aligned}$$

Furthermore, assume that if $\lambda\overline{x}.H(\overline{w})$ is a renaming substitution, then $H \in dom\rho_v$ and $\rho_v(H) = H$. Observe that such an f exists because $g_t < g_v$. We will show that for any $\lambda\overline{x}.H(\overline{w})$, there is a $\longrightarrow$-step which generates an appropriate g_u. In most cases, we only identify which step is applicable;

refer to the algorithm for the details of constructing g_u and ρ_u. Note that if there is a step to create g_u, then $g_u \in \mathbf{R}(a,b)$ by Lemma 6.1.

There are three cases based on the form of H.

1. H is x_i in $\overline{x}$: $\theta_1(f) = \phi_1(\rho_v(f)) = \pi_i = \phi_2(\rho_v(f)) = \theta_2(f)$, so **Delete-variable** is applicable.

2. H is a constant K: $head(\theta_1) = head(\phi_1(\rho_v(f))) = K = head(\phi_2(\rho_v(f))) = head(\theta_2)$, so **Factor-constant** is applicable.

3. H is a free variable (say g): Since we are only interested in finding an isomorphism of g'_v, we can reorder the arguments to g as $g(x_1, \ldots, x_k, w_{k+1}, \ldots, w_n)$ (with corresponding reorderings to $\phi_1(g)$ and $\phi_2(g)$) such that k is the smallest integer for which $w_{k+1} \neq x_{k+1}$.

   If $k = n$, then $\theta_1(f) = \phi_1(\rho_v(f)) = \phi_1(g)$ and by assumption $\theta_1(g) = \phi_1(\rho_v(g)) = \phi_1(g)$. Thus $\theta_1(f) = \theta_1(g)$. Likewise, $\theta_2(f) = \theta_2(g)$. Hence the **Merge** step is applicable.

   If $k < n$, then there are four cases depending upon the form of w_{k+1}:

   (a) w_{k+1} is flexible: this would make g_v redundant, a contradiction.

   (b) $w_{k+1} = x_i$ for $i \leq k$: y_i occurs more than once in both M and N and so a **Factor-abstraction** step is applicable. Pick $\hat{\alpha}$ and $\hat{\beta}$ such that the occurrences of y_i in $\sigma_1(f')$ and $\sigma_2(f')$ match those in $\phi_1(g)$ and $\phi_2(g)$.

   (c) $w_{k+1} = x_i$ for $i > k+1$: y_{k+1} does not occur in either M or N and so a **Delete-abstraction** step is applicable.

   (d) $w_{k+1} = K(\overline{s})$ where K is a constant: because g_v is not cluttered, K must occur in both M and N, thus a **Factor-constant** step is applicable. Again, pick $\hat{\alpha}$ and $\hat{\beta}$ such that the occurrences of K in $\sigma_1(f')$ and $\sigma_2(f')$ match those in $\phi_1(g)$ and $\phi_2(g)$.

□

Theorem 6.4 (Soundness) If $g \in gen(a,b)$, then $g \in \mathrm{MSG}(a,b)$.

Proof By Lemmas 6.1 and 6.2, if $g \in gen(a,b)$ then $g \in \mathbf{R}(a,b)$. g is maximal by Lemma 6.3. □

Theorem 6.5 (Completeness) If g is in $\mathrm{MSG}(a,b)$, then there is a generalization g' in $gen(a,b)$ which is isomorphic to g.

Proof By Observation 4.8, we know that g_0 is less specific than any generalization of a and b, so by Lemma 6.3 we know that for any g there is a sequence of $\longrightarrow$ steps from g_0 to some g' isomorphic to g. This sequence is finite because $<$ is well-founded and $g_1 \longrightarrow g_2$ implies $g_1 < g_2$. □

7 Conclusion

We provide a framework for unsolved problem of generalizing second-order terms. Our solution is based on viewing the structure of terms as a category rather than a partial order. The categorical view allows us to capture *how* one term generalizes another, which is not possible in the conventional structure of complete lattices [14, 16].

Second-order generalization seems eminently useful for generalizing first-order terms in a useful fashion. For instance, A and $A \wedge B$ have the maximal generalization

$$\langle [\pi] : f(\mathtt{A}) \rightarrow \mathtt{A},\ [\lambda x.x \wedge \mathtt{B}] : f(\mathtt{A}) \rightarrow \mathtt{A} \wedge \mathtt{B} \rangle$$

showing that A is replaced by a conjunction in going to $A \wedge B$. This information is lost in the corresponding first-order most specific generalization. Similarly, going to third and higher orders would improve the quality of generalization. More importantly, base terms of higher orders also necessitate going to higher orders. We intend to pursue this in future work.

Acknowledgements

The authors would like to thank John Gray for useful discussions on category theory and the anonymous reviewers for a number of helpful comments. We would also like to thank Paul Taylor for the macros used to generate the diagrams.

References

[1] Michael A. Arbib and Ernest G. Manes. *Arrows, Structures, and Functors: The Categorical Imperative.* Associated Press, New York, 1975.

[2] R. S. Boyer and J. S. Moore. *A Computational Logic.* Academic Press, New York, 1979.

[3] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–58, 1940.

[4] R. L. Constable, *et. al.* *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, New York, 1986.

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

[6] Joseph A. Goguen. What is unification? A categorical view of substitution, equation, and solution.

[7] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. In *Symp. on Logic in Computer Science*, pages 194–204. IEEE, June 1987.

[8] Gérard Huet. Résolution d'Équations dans des langages d'order $1, 2, \ldots, \omega$ (these d'etat), December 1976.

[9] F. William Lawvere. Functional semantics of algebraic theories. In *National Academy of Sciences*, 1963.

[10] Lawrence C. Paulson. Natural deduction as higher-order resolution. *Journal of Logic Programming*, 3:237–258, 1986.

[11] F. Pfenning. Elf: A language for logic definition and verified meta-programming. In *Fourth Ann. Symp. on Logic in Computer Science*, pages 313–322. IEEE, June 1989.

[12] Frank Pfenning. Unification and anti-unification in the calculus of constructions. In *Sixth Annual LICS*, pages 74–85. IEEE, IEEE Computer Society Press, July 1991.

[13] T. Pietrzykowski. A complete mechanization of second-order type theory. *Journal of the ACM*, 20(2):333–365, April 1973.

[14] Gordon D. Plotkin. A note on inductive generalization. In *Machine Intelligence*, volume 5, chapter 8, pages 153–163. Edinburgh Univ. Press, Edinburgh, 1970.

[15] U. S. Reddy. Transformational derivation of programs using the Focus system. *SIGSOFT Software Engineering Notes*, 13(5):163–172, Nov 1988. (Proceedings, ACM SIGSOFT/SIGPLAN Softw. Eng. Symp. on Practical Software Development Environments).

[16] John C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. In *Machine Intelligence*, volume 5, chapter 7, pages 135–151. Edinburgh Univ. Press, Edinburgh, 1970.

Implementing Higher-Order Algebraic Specifications

Jan Heering [1]
Department of Software Technology
CWI
Kruislaan 413
1098 SJ Amsterdam, The Netherlands
jan@cwi.nl

Abstract

Writing algebraic specifications that are to be executed as rewrite systems is similar to functional programming. There are some differences, however. Algebraic specification languages allow left-hand sides of equations to be complex first-order patterns that would not be allowed in functional languages. Functional languages, on the other hand, have powerful higher-order features not offered by algebraic specification languages. Some functional languages combine higher-order functions with linear first-order patterns involving free data type constructors, thus offering a limited (but highly expressive) mixture of functional programming and algebraic specification. A more ambitious integration of the two is obtained by allowing both signatures and equations in algebraic specifications to be higher-order. Operational experiments with such higher-order algebraic specifications can be performed by translating them to λProlog, an extension of Prolog to polymorphically typed λ-terms based on higher-order unification.

1 Introduction

1.1 Higher-order algebraic specifications

Conventional algebraic data type specifications consist of a first-order signature and a set of equations. Equations may contain first-order variables, which are implicitly or explicitly universally quantified. The signature defines the abstract syntax of a language of terms whose semantics is given by the equations. Such specifications are usually implemented by interpreting them as (first-order) term rewriting systems (see the survey by Klop [13]). Each equation is interpreted as a left-to-right rewrite rule and the resulting rewrite system is used to evaluate terms by reducing them to normal form (if any). The annoying fact that this asymmetric interpretation of inherently symmetric equations may lead to rewrite systems that are incomplete with respect to equational deduction from the original specification does not concern us here.

Writing algebraic specifications that are to be executed as rewrite systems is similar to functional programming. There are some differences, however. Algebraic specification languages allow left-

[1]Supported in part by the European Communities under ESPRIT project 2177 (Generation of Interactive Programming Environments II—GIPE II).

hand sides of equations to be complex first-order patterns that would not be allowed in functional languages. Functional languages, on the other hand, have powerful higher-order features not offered by algebraic specification languages.

Some functional languages (e.g., Hope [1, 2]) combine higher-order functions with linear first-order patterns involving free data type constructors, thus offering a limited (but highly expressive) mixture of functional programming and algebraic specification. A more ambitious integration of the two is obtained by allowing both signatures and equations in algebraic specifications to be higher-order. The higher-order signature defines the abstract syntax of a language of typed λ-terms whose semantics is given by the equations. Parsaye-Ghomi has been one of the first to study this approach [21].

More recently, Jouannaud and Okada [12] have advocated the development and implementation of higher-order algebraic specification languages and, having frequently felt the need for higher-order equations in algebraic specifications ourselves, we thought it would be interesting to be able to perform operational experiments with them. Higher-order term rewriting requires, first of all, higher-order matching, which is the special case of higher-order unification in which one of the terms involved does not contain free variables. Two readily available systems incorporating higher-order unification are λProlog [20], an extension of Prolog to typed λ-terms, and the generic theorem prover Isabelle [22]. Since we had some experience with schemes for translating first-order algebraic specifications to Prolog (see the surveys by Drosten [7] and Bouma and Walters [4]), we chose λProlog as our target system.

It would be nice if the notion of initial algebra specification, which has unequivocal meaning in the first-order case [16], had an equally unequivocal higher-order analogue. This does not seem to be the case, however, since it depends on the precise notion of higher-order model one prefers. Meinke and Möller, for instance, assume models to be extensional higher-order algebras [15, 18], and Meinke shows that in this setting higher-order initial algebra specification is strictly more powerful than its first-order counterpart [14]. Poigné, on the other hand, considers both extensional and intensional models [23]. Although these questions are beyond our present scope, the precise notion of initial algebra semantics adopted affects the degree of incompleteness of our implementation scheme.

1.2 Higher-order term rewriting

Higher-order term rewriting, the mechanism we use to execute higher-order algebraic specifications, is more powerful, but also less manageable than its first-order counterpart. The following examples illustrate some of its possibilities and problems.

I. Consider the signature

sorts $s, bool$
functions
$a : s$
$f, g : s \rightarrow s$
$if : bool \times s \times s \rightarrow s$

variables

$X, Y : s$

$F : s \rightarrow s$ (second-order variable)

$B, B' : bool$

and the second-order equation

$$if(B, F(X), F(Y)) = F(if(B, X, Y)). \tag{1}$$

The left-hand side of (1) matches

$$if(B', g(f(a)), g(f(f(a))))$$

in three different ways, namely, for

$$\begin{array}{llll} F = \lambda V.g(f(V)) & X = a & Y = f(a) & B = B' \\ F = \lambda V.g(V) & X = f(a) & Y = f(f(a)) & B = B' \\ F = \lambda V.V & X = g(f(a)) & Y = g(f(f(a))) & B = B'. \end{array}$$

Thus, whereas a first-order match has at most a single solution, a higher-order match may have many. It may even have solutions that leave some of the variables in the left-hand side of the rewrite rule uninstantiated, something that cannot happen in the first-order case either. For instance, the left-hand side of (1) matches

$$if(B', a, a)$$

for

$$\begin{array}{llll} F = \lambda V.a & X = X & Y = Y & B = B' \\ F = \lambda V.V & X = a & Y = a & B = B'. \end{array}$$

The first solution leaves X and Y uninstantiated. If (1) is interpreted as a left-to-right rewrite rule, this is no problem since both variables are eliminated by β-reduction after substitution of the solution in the right-hand side:

$$if(B', a, a) \xrightarrow{(1)} (\lambda V.a)(if(B', X, Y)) \xrightarrow{(\beta)} a.$$

A solution instantiating F to $\lambda V.V$ exists for any if-term and is, at least in this case, algebraically harmless. The danger of non-termination it entails can be averted by adopting a parallel reduction strategy treating all solutions on an equal basis, or by a simple loop check. For reasons of efficiency we have chosen the latter alternative.

II. Consider the second-order equations

$$\begin{aligned} map(F, nil) &= nil & (2) \\ map(F, cons(X, L)) &= cons(F(X), map(F, L)) & (3) \\ map(\lambda V.V, L) &= L & (4) \\ map(F, map(G, L)) &= map(\lambda V.F(G(V)), L) & (5) \end{aligned}$$

with the signature from example (I) plus the additional declarations

sort *lst*
functions
  $nil : lst$
  $cons : s \times lst \rightarrow lst$
  $map : (s \rightarrow s) \times lst \rightarrow lst$
variables
  $X, V : s$
  $L : lst$
  $F, G : s \rightarrow s$ (second-order variables).

Equations (2) and (3) define the *map*-function for the basic list constructor cases. They could have been written in virtually the same way in Hope [1, Chapter 6]. Equations (4) and (5) are plausible identities for the *map*-function. These would not be allowed in Hope since their left-hand sides involve arguments $\lambda V.V$ and $map(G, L)$ which are not constructor terms. From the viewpoint of higher-order matching these are harmless, however.

III. Although it did not happen in example (1), variables in the left-hand side of a higher-order rewrite rule that are left uninstantiated after matching may enter the reduct. We borrow the following example from Nipkow's paper on higher-order critical pairs [19]. The rewrite rule

$$f(g(F(X), F(a))) \rightarrow f(X)$$

can be applied to the term $f(g(a, a))$ in two ways, one of which instantiates F to $\lambda V.a$ and leaves X uninstantiated, thus yielding the result $f(X)$.
To get rid of this problem and to eliminate ambiguous rules such as (1), Nipkow (following Miller [17]) restricts left-hand sides of rules to so-called *higher-order patterns* (HOPs). A HOP is a term in β-normal form such that each free variable occurring in it is applied only to (zero or more) terms that are η-equivalent to distinct bound variables. The left-hand sides of equations (2)–(5) are HOPs, but the left-hand side of (1) is not since it contains a free variable F whose argument X is not a bound variable. Jouannaud and Okada's notion of *general schema* [12, Section 4.4] does not include equation (1) either.
To leave as much room for experiment as possible, we do not impose any a priori restriction, but equations that may cause uninstantiated variables to enter the reduct are not necessarily treated correctly by our λProlog code and should be avoided.

IV. Whereas first-order term rewriting requires subterm matching, higher-order rewriting can do without explicit subterm lookup if each equation $t_1 = t_2$ is extended to $H(t_1) = H(t_2)$ with H a polymorphic higher-order variable not free in t_1 or t_2. In this case, higher-order matching of the extended left-hand side with the full input term performs the subterm lookup implicitly. Like before, useless instantiations of H to $\lambda X.s$, where s does not contain X, can be rejected by a simple loop check. The matching strategy used does not matter as long as the rewrite system is confluent and terminating (apart from the trivial loops caught by the loop check). This approach is used in Section 2.1. Tactics for higher-order rewriting are discussed by Felty [8].

1.3 λProlog

λProlog is an extension of Prolog to typed λ-terms [20]. Basically, the functions declared in a λProlog program generate a domain of polymorphically typed λ-terms, and polymorphic higher-order unification takes the place of first-order unification in the proof procedure.

Since λ-terms may be subject to α-, β-, and η-reduction, the term domain underlying a λProlog program is not purely syntactic. Furthermore, unlike first-order unification, higher-order unification is neither decidable nor unitary. As a consequence, in λProlog backtracking to an alternative unifier of the same pair of terms may occur and the search for a higher-order unifier may go on forever.

Higher-order matching, the special case of higher-order unification we need, was conjectured to be decidable in the simply typed case (no polymorphism) by Huet [11], but this is still an open problem. The third-order case was recently shown to be decidable by Dowek [5]. On the other hand, Dowek also showed that strongly polymorphic higher-order matching is undecidable [6]. λProlog supports ML-style polymorphism, so we included it in our notion of higher-order algebraic specification as well, in accordance with Parsaye-Ghomi's original proposal [21]. As far as we know, the "intermediate" case of higher-order matching in combination with ML-style polymorphism has not yet been settled, so it may still turn out to be decidable. In the version of λProlog we used[2] the implementation of polymorphic higher-order unification was incomplete and this caused some problems. These will be explained in due course. Examples of higher-order matches with multiple solutions, none of them subsumed by any of the other ones, were given in Section 1.2. In our λProlog code, backtracking to an alternative solution may occur as a result of loop checking.

This rudimentary knowledge of λProlog in combination with a basic understanding of Prolog (see, for instance, Bratko's book [3]) suffices to understand the next section.

2 Translating higher-order algebraic specifications to λProlog

2.1 A very simple scheme

Consider the following higher-order algebraic specification:

module N
sorts $nat, bool, lst(A)$
functions

$zero : nat$
$succ : nat \rightarrow nat$
$add : nat \times nat \rightarrow nat$
$t, f : bool$
$if : bool \times A \times A \rightarrow A$
$nil : lst(A)$
$cons : A \times lst(A) \rightarrow lst(A)$
$compose : (B \rightarrow C) \times (A \rightarrow B) \rightarrow A \rightarrow C$
$map : (A \rightarrow B) \times lst(A) \rightarrow lst(B)$

[2] Version 2.7 (October 1988). It was obtained by anonymous *ftp* from *duke.cs.duke.edu*.

equations

$$
\begin{aligned}
add(X, zero) &= X & (6)\\
add(X, succ(Y)) &= succ(add(X, Y)) & (7)\\
if(t, X, Y) &= X & (8)\\
if(f, X, Y) &= Y & (9)\\
if(B, F(X), F(Y)) &= F(if(B, X, Y)) & (10)\\
compose(F, G) &= \lambda X.F(G(X)) & (11)\\
map(F, nil) &= nil & (12)\\
map(F, cons(X, L)) &= cons(F(X), map(F, L)) & (13)\\
map(\lambda V.V, L) &= L & (14)\\
map(F, map(G, L)) &= map(compose(F, G), L) & (15)
\end{aligned}
$$

Identifiers whose first character is a capital letter are variables. Their type is not declared explicitly (although it might have been), but is determined by the context in which they occur. For instance, X has type *nat* in (6), but polymorphic type A (with A a type variable) in (8).

In addition to the two carriers corresponding to sorts *nat* and *bool*, the higher-order initial algebra of N has an infinite number of first-order carriers corresponding to $lst(\tau)$ for any monotype τ. In particular, τ may be a functional monotype such as $nat \rightarrow nat$ or another *lst*-monotype. The higher-order carriers (function spaces) of the initial algebra consist of the appropriately typed functions definable in terms of the signature of N.

Equations (10) and (12)–(15) are polymorphic versions of (1) and (2)–(5) respectively. Equation (11) defines functional composition. Equations (10), (14), and (15) merit special attention. These are the ones that are allowed in the higher-order algebraic framework, but not in Hope. As was pointed out in Section 1.2, the left-hand side of (10) is highly non-deterministic. The left-hand sides of (14) and (15) are HOPs of a simple kind, but not constructor cases.

Using the scheme outlined in example (IV) of Section 1.2, we translate N to the following λProlog module:

```
module lpN.

kind nat      type.
kind bool     type.
kind lst      type -> type.

type zero     nat.
type succ     nat -> nat.
type add      nat -> nat -> nat.
type t        bool.
type f        bool.
type if       bool -> A -> A -> A.
```

```
type nil     (lst A).
type cons    A -> (lst A) -> (lst A).
type map     (A -> B) -> (lst A) -> (lst B).
type compose (B -> C) -> (A -> B) -> A -> C.

type reduce  A -> A -> o.
type extrule A -> A -> o.

extrule (H (add X zero))          (H X).                             %%% (6a)
extrule (H (add X (succ Y)))      (H (succ (add X Y))).              %%% (7a)
extrule (H (if t X Y))            (H X).                             %%% (8a)
extrule (H (if f X Y))            (H Y).                             %%% (9a)
extrule (H (if B (F X) (F Y)))    (H (F (if B X Y))).                %%% (10a)
extrule (H (compose F G))         (H (X \ (F (G X)))).               %%% (11a)
extrule (H (map F nil))           (H nil).                           %%% (12a)
extrule (H (map F (cons X L)))    (H (cons (F X) (map F L))).        %%% (13a)
extrule (H (map X \ X L))         (H L).                             %%% (14a)
extrule (H (map F (map G L)))     (H (map (compose F G) L)).         %%% (15a)

reduce X Y :- extrule X Z,
                not(X = Z), %%% loop check - X,Z ground
                  reduce Z Y.                                        %%% (16)
reduce X X.                                                          %%% (17)
```

Arguments of predicates are separated by spaces rather than commas in λProlog, and the argument list of a predicate is not delimited by brackets. The syntax of λ-terms is similar to that of Lisp. Every predicate or function is at most unary, so larger arities have to be reduced to arity 1 by currying, that is, by replacing types $s_1 \times \cdots \times s_k \to s_0$ in the algebraic specification with types `s1 -> ... -> sk -> s0` in λProlog. As usual, the type constructor `->` is right-associative. Predicates always have type `... -> o`.

Kind declarations are used to introduce type constructors. The three `kind` declarations in the first lines of `lpN` introduce the zero-adic type constructors `nat` and `bool`, and the monadic type constructor `lst`. These correspond to the sorts *nat*, *bool*, and $lst(A)$ of N. Thus, apart from the declarations of the auxiliary predicates `extrule` and `reduce`, the correspondence between the signatures of N and `lpN` is straightforward. The translation of equations is equally straightforward. Put in the context of a new higher-order variable `H`, the left- and right-hand side of an equation become the first and second argument of the corresponding `extrule` fact. Note that $\lambda X. \cdots$ in the right-hand side of (11) becomes `(X \ ...` in λProlog. In addition to the `extrule` facts corresponding to the equations of N, the body of `lpN` consists of the clauses `(16)` and `(17)` for `reduce`. These are independent of N.

The normal form of a term t in the term language defined by the signature of N is obtained by submitting to `lpN` the question

```
?- reduce tt NF.
```

where **tt** is the corresponding term in the term language of **lpN**. Since free variables in t (if any) should not be instantiated during rewriting, they do not correspond to λProlog variables in **tt**, but are modelled by generic constants (simulated variables) **x**, **y**, ... in the following examples. Thus, even if t contains free variables, **tt** is a ground term.

Rewriting proceeds as follows. The **reduce** predicate attempts to apply **extrule** and, if successful, calls itself recursively on the reduct after performing the loop check **not(X = Z)**, where **not** is the negation-as-failure predicate and **=** denotes higher-order unification. The loop check rejects algebraically correct but operationally useless matches (cf. Section 1.2, examples (I) and (IV)). When it is evaluated, the values of both **X** and **Z** are ground terms because (i) the translated input term **tt** is always ground, and (ii) the equations are assumed to be such that their interpretation as left-to-right rewrite rules does not cause uninstantiated variables to enter the reduct (cf. Section 1.2, example (III)).

The rewrite strategy of **lpN** is determined primarily by the fact that β-reduction is a built-in rewrite rule that is performed implicitly by λProlog during unification, and by the order of the **extrule** facts. Redexes for rule r_m are reduced before redexes for rule r_n if $m < n$. The redex selection strategy for each individual rule is determined by λProlog's higher-order unification strategy. The latter can be influenced to some extent by the setting of the **projfirst** switch of the λProlog system. If set to on, the higher-order unification machinery prefers projection over imitation. This reduces the amount of backtracking caused by imitative solutions that are rejected by the loop check, and promotes the simultaneous reduction of syntactically identical redexes.

We reproduce a short sample run of the λProlog system using **lpN**:

```
?- use lpN.
lpN
yes

?- switch projfirst on.          %%% slightly more efficient in this
yes                              %%%  application than projfirst off

?- switch tvw off.               %%% no type variable instantiation warnings
 yes

?- reduce (if y (cons f nil) (cons t nil)) NF.
                                 %%% y is a generic constant - see above
NF = cons (if y f t) nil .
 yes

?- reduce (if y (add (succ zero) (succ zero)) (succ (succ zero))) NF.
                                 %%% y is a generic constant - see above
NF = succ (succ zero) .
 yes

?- reduce (if y (if y1 x0 x1) (if y1 x2 x1)) NF.
```

```
NF = if y1 (if y x0 x2) x1 .   %%% see [9, Section 3.3]
 yes

?- reduce ((compose (X \ (add X X)) (X \ (add X X))) (succ zero)) NF.

NF = succ (succ (succ (succ zero))) .
 yes

?- reduce (map (X \ (add X X)) (cons zero (cons (succ zero) nil))) NF.

NF = cons zero (cons (succ (succ zero)) nil) .
 yes

?- reduce (map (X \ (compose succ X)) (cons succ nil)) NF.

NF = cons Var1612 \ (succ (succ Var1612)) nil .
 yes

?- reduce (map (X \ zero) (map succ l)) NF.
                               %%% l is a generic constant - see above
NF = map (Var347 \ zero) l .
 yes

?- reduce (Y \ (add Y zero)) NF.

NF = Y \ (add Y zero) .        %%% no rewriting under abstraction;
 yes                           %%% first argument of (6a) does not
                               %%% match - see Section 2.3

?- reduce (if y succ succ) NF.

NF = if y succ succ .          %%% NF = succ expected - see below
 yes
```

The last example is not reduced properly because the implementation of polymorphic higher-order unification in the version of λProlog we used was incomplete. When matching `if y succ succ` with the left-hand side of (10a), the polytype `A1 -> nat -> nat` initially inferred for `H` is never instantiated to `(nat -> nat) -> nat -> nat`. The reason is that the system limits `A1` to "primitive" types to keep the search space within bounds. It is interesting to see how the matching behaves in this case:

```
?- switch tvw on.              %%% give type variable instantiation warnings
 yes
```

```
?- switch printtypes on.       %%% print types of terms
 yes

?- if y succ succ = (H (if B (F X) (F Y))).
                              %%% "=" denotes higher-order unification
Trying to project on an argument with type
       A1
Do you want to go on? (y/n)y
Assuming for the moment that target type is primitive

H = Var24 : A1 \ Var25 : nat \
     (if y Var26 : nat \ (succ Var26) Var27 : nat \ (succ Var27) Var25)
B = B : bool
X = X : A1
F = F : A1 -> A2
Y = Y : A1 ;

 no
```

The only solution found leaves all variables in the left-hand side of `(10a)` except `H` uninstantiated and is rejected by the loop check. The expected solution is found if the more precise type **`(nat -> nat) -> nat -> nat`** is associated with H in an *ad hoc* fashion:

```
?- if y succ succ = (H : (nat -> nat) -> nat -> nat (if B (F X) (F Y))).

H = Var26 : nat -> nat \ Var27 : nat \ (Var26 Var27)
B = y
X = X : A1
F = Var28 : A1 \ Var29 : nat \ (succ Var29)
Y = Y : A1 ;

H = Var54 : nat -> nat \ Var55 : nat \
     (if y Var56 : nat \ (succ Var56) Var57 : nat \ (succ Var57) Var55)
B = B : bool
X = X : A1
F = F : A1 -> nat -> nat
Y = Y : A1 ;

 no
```

The first solution yields the expected reduct when substituted in the right-hand side of `(10a)`. The second solution is a more precisely typed version of the useless one found previously.

Finally, we give an example showing that `lpN` is not confluent for terms containing free variables. An alternative normal form can be obtained by backtracking. Note that `lpN` does not do this automatically.

```
?- reduce (if y (add x zero) (add x (succ zero))) NF.
                                %%% x and y are generic constants

NF = if y x (succ x) ;          %%% first normal form

NF = if y x (succ (add x zero)) ;
                                %%% not a normal form
  ...

NF = add x (if y zero (succ zero)) ;
                                %%% second normal form
  ...

 no
```

The general translation scheme should be clear from `lpN`. The auxiliary names **`reduce`**, **`extrule`** and `H` should be chosen carefully to avoid clashes with user-defined names. Similarly, overloading of names that have a predefined meaning in λProlog (`true`, `false`, `list`, ...) should be avoided. Apart from the above-mentioned incompleteness problem and the possible non-termination of higher-order matching (which we have not encountered so far), the scheme is correct for higher-order rewrite systems that do not introduce new variables in the reduct, and that are terminating with the simple loop check shown as well as confluent. For rewrite systems lacking the latter property, the input term may have other normal forms besides the one computed.

2.2 Improving efficiency by adding specialized code

Some efficiency can be gained by combining the above method with one of the first-order schemes discussed in [4, 7]. To illustrate the general idea, we take Drosten and Ehrich's first-order scheme. In this case the λProlog code generated for N becomes:

```
module lpN2.

import lpN.       %%% see Section 2.1

type reduce2       A -> A -> o.
type analyze       A -> A -> o.
type prenormalize A -> A -> o.
type rule          A -> A -> o.
```

```
rule (add X zero)            X.                                    %%% (6b)
rule (add X (succ Y))        (succ (add X Y)).                     %%% (7b)
rule (if t X Y)              X.                                    %%% (8b)
rule (if f X Y)              Y.                                    %%% (9b)
rule (if B (F X) (F Y))      (F (if B X Y)).                       %%% (10b)
rule (compose F G)           (X \ (F (G X))).                      %%% (11b)
rule (map F nil)             nil.                                  %%% (12b)
rule (map F (cons X L))      (cons (F X) (map F L)).               %%% (13b)
rule (map X \ X L)           L.                                    %%% (14b)
rule (map F (map G L))       (map (compose F G) L).                %%% (15b)

analyze (succ I1) K        :- analyze I1 K1,
                               prenormalize (succ K1) K.           %%% (18)
analyze (add I1 I2) K      :- analyze I1 K1, analyze I2 K2,
                               prenormalize (add K1 K2) K.         %%% (19)
analyze (if I1 I2 I3) K    :- analyze I1 K1, analyze I2 K2, analyze I3 K3,
                                prenormalize (if K1 K2 K3) K.      %%% (20)
analyze (compose I1 I2) K :- analyze I1 K1, analyze I2 K2,
                               prenormalize (compose K1 K2) K.     %%% (21)
analyze (cons I1 I2) K     :- analyze I1 K1, analyze I2 K2,
                               prenormalize (cons K1 K2) K.        %%% (22)
analyze (map I1 I2) K      :- analyze I1 K1, analyze I2 K2,
                               prenormalize (map K1 K2) K.         %%% (23)

analyze X K                :- prenormalize X K.                    %%% (24)

prenormalize X Y           :- rule X Z,
                               not(X = Z),  %%% loop check
                                analyze Z Y.                       %%% (25)
prenormalize X X.                                                  %%% (26)

reduce2 X Y                :- analyze X Z, reduce Z Y.             %%% (27)
                                      %%% reduce is defined in lpN
```

`lpN2` extends `lpN` with code that is very similar to the Prolog code that would be generated by Drosten and Ehrich's scheme for N had it been a first-order specification. For each p-ary function f in the signature of N ($p >= 1$), `lpN2` contains a clause

```
analyze (f I1 ... Ip) K    :- analyze I1 K1, ... , analyze Ip Kp,
                               prenormalize (f K1 ...  Kp).
```

Clause `(24)` catches everything not matched by the first argument of the preceding `analyze` cases. The facts `(6b)`–`(15b)` correspond directly to the equations (6)–(15). Clause `(27)` links the new code to the old code imported from `lpN`. The clauses `(24)`–`(27)` are independent of N.

The normal form of a term t in the term language defined by the signature of N is obtained by submitting to `lpN2` the question

```
?- reduce2 tt NF.
```

where `tt` is the corresponding term in the term language of `lpN2` (which is the same as that of `lpN`). Like before, free variables in t have to be replaced by generic constants in `tt` (see Section 2.1).

On the examples we tried, `lpN2` was from 1 to 5 times faster than `lpN`. It may actually be slightly slower if `analyze` is unable to perform any reductions. Consider, for instance, the term

```
(compose succ succ) zero.
```

The first argument of `(21)` does not match (its type is not even compatible), so the work done by `analyze` is wasted and the reduction to `succ (succ zero)` is performed by `reduce` using `(11a)` with

```
H = Var : nat -> nat \ (Var zero)
F = succ
G = succ .
```

On the other hand, the reduction of

```
map (X \ (compose succ X)) (cons succ nil)
```

to

```
cons Var \ (succ (succ Var)) nil
```

is speeded up by a factor of 5. Whereas `lpN` spends a large amount of time on useless matches, `lpN2` performs the reduction in a highly deterministic manner using `analyze`.

2.3 Reduction under abstraction and partial evaluation

Evaluation of programs whose input values are only partially given is called *partial evaluation* (see the annotated bibliography [24]). In the setting of first-order algebraic specification, partial evaluation corresponds to reduction of first-order terms containing free variables [9]. In Section 2.1 we gave several examples of this in the setting of higher-order algebraic specification. In fact, the equations

$$\begin{aligned} if(B, F(X), F(Y)) &= F(if(B, X, Y)) \\ map(\lambda V.V, L) &= L \\ map(F, map(G, L)) &= map(\lambda V.F(G(V)), L), \end{aligned}$$

which played a role in some of the examples, are not needed for ordinary evaluation, but may be useful for partial evaluation. Needless to say, more equations of this kind could have been added to the specification N.

In the higher-order setting, partial evaluation not only corresponds to reduction of open terms, however, but also to reduction under abstraction. The two are related by the *abstraction rule*

$$\frac{\vdash t_1 = t_2}{\vdash \lambda X.t_1 = \lambda X.t_2},$$

which has no analogue in the first-order case. For instance, according to the abstraction rule one would expect an implementation of N to reduce $\lambda Y.add(Y, zero)$ to $\lambda Y{:}nat.Y$, since $add(Y, zero)$ reduces to $Y : nat$ by equation (6). The two implementations discussed so far do not do this, however:

```
?- reduce (add y zero) NF.      %%% y is a generic constant

NF = y .                        %%% OK, but ...
 yes

?- reduce (Y \ (add Y zero)) NF.

NF = Y \ (add Y zero) .         %%% first argument of (6a) does not match
 yes

?- reduce2 (Y \ (add Y zero)) NF.

NF = Y \ (add Y zero) .         %%% the analyze-predicate of lpN2 does not descend
 yes                            %%% into abstractions
```

We note that the fact that `lpN` and `lpN2` do not perform reduction under abstraction is in accordance with common functional programming practice.

Picking up an abstraction in the style of `lpN` would require higher-order matching with

$$H(\lambda X.U(X)),$$

but the incomplete instantiation of type variables during unification mentioned in Section 2.1 precludes this approach. Instead, we add a case to the definition of the `analyze`-predicate in `lpN2` just before (24):

```
analyze (X \ (U X)) (X \ (V X)) :- pi C \ (reduce2 (U C) (V C)).          %%% (24-)
```

When it recognizes an abstraction `(X \ (U X))`, `analyze` uses λProlog's built-in `pi`-predicate to convert it to a generic instance `(U C)` in the universal goal `reduce2 (U C) (V C)`. (Universal goals in λProlog are discussed by Nadathur and Miller in [20, pp. 817–818].) After normalization by `reduce2`, the resulting normal form `(V C)` is turned into an abstraction `(X \ (V X))`. For instance,

```
?- reduce2 (L \ (map (X \ (add X zero)) L)) NF.
                                %%% application of (24-), (23), (6b), and (14b)
                                %%% yields the identity function of type
                                %%% lst nat -> lst nat:
NF = Var335 : lst nat \ Var335 .
 yes
```

Like (24)-(27), clause (24-) is independent of N.

We conclude this section by pointing out that reduction of polymorphic abstractions is prone to divergence. For instance, reduction of the identity function (X : A \ X) of polymorphic type A -> A leads to an infinite loop. Clause (24-) remains applicable after each generic instantiation.

3 Further work

From a logical viewpoint, higher-order algebraic specification constitutes a natural integration of first-order algebraic specification and higher-order functional programming. We intend to perform further experiments with it using the implementation schemes discussed in this paper and perhaps more efficient ones still to be developed (see, for instance, [10]). Since polymorphic typing has been the main source of problems so far, it requires special attention.

Acknowledgement

A suggestion by one of the reviewers of the λProlog Workshop to use the pi-predicate to do rewriting under abstraction has been very helpful and was incorporated in Section 2.3.

References

[1] R. Bailey, *Functional Programming with Hope* (Ellis Horwood, 1990).

[2] R. Burstall, D. MacQueen, and D. Sannella, Hope: an experimental applicative language, in: *Conference Record of the 1980 Lisp Conference*, Stanford, 1980, 136–143.

[3] I. Bratko, *Prolog Programming for Artificial Intelligence* (Addison-Wesley, 1986).

[4] L.G. Bouma and H.R. Walters, Implementing algebraic specifications, in: J.A. Bergstra, J. Heering, and P. Klint, eds., *Algebraic Specification* (ACM Press/Addison-Wesley, 1989) 199–282.

[5] G. Dowek, Third-order matching is decidable. Rapport de Recherche, INRIA-Rocquencourt, 1991.

[6] G. Dowek, The undecidability of pattern matching in calculi where primitive recursive functions are representable. Rapport de Recherche, INRIA-Rocquencourt, 1991.

[7] K. Drosten, Translating algebraic specifications to Prolog programs: a comparative study, in: J. Grabowski, P. Lescanne, and W. Wechler, eds., *Algebraic and Logic Programming*, Lecture Notes in Computer Science, Vol. 343 (Springer-Verlag, 1988) 137–146.

[8] A. Felty, A logic programming approach to implementing higher-order term rewriting, in: L.-H. Eriksson, L. Hallnäs, and P. Schroeder-Heister, eds., *Extensions of Logic Programming*, Lecture Notes in Artificial Intelligence, Vol. 596 (Springer-Verlag, 1991) 135–161.

[9] J. Heering, Partial evaluation and ω-completeness of algebraic specifications, *Theoretical Computer Science*, **43** (1986) 149–167.

[10] J. Heering, Second-order algebraic specification of static semantics, Report, CWI, Amsterdam, in preparation.

[11] G. Huet, Résolution d'équations dans les langages d'ordre $1,2,\ldots,\omega$, Thèse de Doctorat d'Etat, Université de Paris-VII, 1976.

[12] J.-P. Jouannaud and M. Okada, A computation model for executable higher-order algebraic specification languages, in: *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science* (IEEE Computer Society Press, 1991) 350–361.

[13] J.W. Klop, Term rewriting systems, in: S. Abramsky, D. Gabbay, and T. Maibaum, eds., *Handbook of Logic in Computer Science*, Vol. II (Oxford University Press).

[14] K. Meinke, A recursive second order initial algebra specification of primitive recursion, Report CSR 8-91, Computer Science Division, Department of Mathematics and Computer Science, University College of Swansea, June 1991.

[15] K. Meinke, Universal algebra in higher types, *Theoretical Computer Science*, **100** (1992) 385–417.

[16] J. Meseguer and J.A. Goguen, Initiality, induction, and computability, in: M. Nivat and J.C. Reynolds, eds., *Algebraic Methods in Semantics* (Cambridge University Press, 1985) 459–541.

[17] D. Miller, A logic programming language with lambda-abstraction, function variables, and simple unification, in: P. Schroeder-Heister, ed., *Extensions of Logic Programming*, Lecture Notes in Artificial Intelligence, Vol. 475 (Springer-Verlag, 1991) 253–281.

[18] B. Möller, Algebraic specification with higher-order operators, in: L.G.L.T. Meertens, ed., *Program Specification and Transformation* (North-Holland/IFIP, 1987) 367–398.

[19] T. Nipkow, Higher-order critical pairs, in: *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science* (IEEE Computer Society Press, 1991) 342–349.

[20] G. Nadathur and D. Miller, An overview of λProlog, in: R.A. Kowalsi and K.A. Bowen, eds., *Logic Programming—Proceedings of the Fifth International Conference and Symposium*, Vol. 1 (The MIT Press, 1988) 810–827.

[21] K. Parsaye-Ghomi, Higher-order abstract data types, Report CSD-820112, Computer Science Department, University of California, Los Angeles, January 1982.

[22] L.C. Paulson and T. Nipkow, Isabelle tutorial and user's manual, Technical Report No. 189, Computer Laboratory, University of Cambridge, January 1990.

[23] A. Poigné, On specifications, theories, and models with higher types, *Information & Control*, **68** (1986) 1–46.

[24] P. Sestoft and A.V. Zamulin, eds., Annotated bibliography on partial evaluation and mixed computation, *New Generation Computing*, **6** (1988) 309–354.

Lolli: An Extension of λProlog with Linear Logic Context Management

Joshua S. Hodas [1]
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104-6839 USA
hodas@saul.cis.upenn.edu

Introduction

The announcement for this workshop began with a passage about the utility of higher-order hereditary Harrop formulas for many applications, and the very existence of the workshop is a partial correctness proof of the passage. Nevertheless, there are applications for which the intuitionistic management of proof contexts (or, concretely, program databases) provided by λProlog has been unable to provide natural, logical solutions. Many such problems, such as how to program the Prolog **bag_of** predicate — which would require a way of augmenting the database such that the changes survive a failure — seem unlikely to yield to logical analysis in any system related to hereditary Harrop formulas. Others, however, can be addressed by relatively simple modifications of the logic underlying λProlog.

In 1990 two problems motivated Dale Miller and me to examine the possibility of designing a logic programming language based on a fragment of Girard's linear logic [2] similar to the hereditary Harrop formula fragment of intuitionistic logic.

The first problem involved representing the notion of mutable object state within logic programming [3]. While it is simple to use representative predicates to store the state of an object in the database (or proof context), it is not possible to model the modification of state, since the only change to the database allowed in λProlog is that of stack-like augmentation through the use of implications in goals. Thus, if the state of a switch is stored using the predicates *off* and *on*, and the program Γ includes the (slightly) higher-order clauses:

$$\Gamma = \begin{cases} \forall G.[toggle(G) \subset (on \wedge (off \supset G))] \\ \forall G.[toggle(G) \subset (off \wedge (on \supset G))] \end{cases}$$

[1] The author has been funded by ONR N00014-88-K-0633, NSF CCR-91-02753, and DARPA N00014-85-K-0018 through the University of Pennsylvania.

$$\frac{}{\Gamma; A \longrightarrow A}\ \text{identity} \qquad \frac{}{\Gamma; \Delta \longrightarrow \top}\ \top_R \qquad \frac{}{\Gamma; \emptyset \longrightarrow 1}\ 1_R \qquad \frac{\Gamma, B; \Delta, B \longrightarrow C}{\Gamma, B; \Delta \longrightarrow C}\ \textit{absorb}$$

$$\frac{\Gamma; \Delta, B_i \longrightarrow C}{\Gamma; \Delta, B_1 \mathbin{\&} B_2 \longrightarrow C}\ \&_{Li} \qquad \frac{\Gamma; \Delta \longrightarrow B \qquad \Gamma; \Delta \longrightarrow C}{\Gamma; \Delta \longrightarrow B \mathbin{\&} C}\ \&_R$$

$$\frac{\Gamma; \Delta_1 \longrightarrow B \qquad \Gamma; \Delta_2, C \longrightarrow E}{\Gamma; \Delta_1, \Delta_2, B \multimap C \longrightarrow E}\ \multimap_L \qquad \frac{\Gamma; \Delta, B \longrightarrow C}{\Gamma; \Delta \longrightarrow B \multimap C}\ \multimap_R$$

$$\frac{\Gamma; \emptyset \longrightarrow B \qquad \Gamma; \Delta, C \longrightarrow E}{\Gamma; \Delta, B \Rightarrow C \longrightarrow E}\ \Rightarrow_L \qquad \frac{\Gamma, B; \Delta \longrightarrow C}{\Gamma; \Delta \longrightarrow B \Rightarrow C}\ \Rightarrow_R$$

$$\frac{\Gamma; \Delta_1 \longrightarrow B \qquad \Gamma; \Delta_2 \longrightarrow C}{\Gamma; \Delta_1, \Delta_2 \longrightarrow B \otimes C}\ \otimes_R \qquad \frac{\Gamma; \emptyset \longrightarrow G}{\Gamma; \emptyset \longrightarrow !G}\ !G$$

$$\frac{\Gamma; \Delta \longrightarrow B[t/x]}{\Gamma; \Delta \longrightarrow \exists x.B}\ \exists_R \qquad \frac{\Gamma; \Delta \longrightarrow B_i}{\Gamma; \Delta \longrightarrow B_1 \oplus B_2}\ \oplus_{Ri}$$

$$\frac{\Gamma; \Delta, B[t/x] \longrightarrow C}{\Gamma; \Delta, \forall x.B \longrightarrow C}\ \forall_L \qquad \frac{\Gamma; \Delta \longrightarrow B[y/x]}{\Gamma; \Delta \longrightarrow \forall x.B}\ \forall_R$$

provided that y is not free in the lower sequent.

Figure 1: A proof system for the connectives *top*, 1, $\&$, $\multimap$, $\Rightarrow$, $!$, $\otimes$, $\oplus$, $\forall$, and $\exists$.

then the proof of the goal $\mathit{off} \supset \mathit{toggle}(G)$ might proceed as follows:

$$\cfrac{\cfrac{\cfrac{\cfrac{}{\Gamma, \mathit{off} \longrightarrow \mathit{off}} \quad \cfrac{\begin{matrix}\vdots \\ \Gamma, \mathit{off}, \mathit{on} \longrightarrow G\end{matrix}}{\Gamma, \mathit{off} \longrightarrow \mathit{on} \supset G}\ \multimap_R}{\Gamma, \mathit{off} \longrightarrow \mathit{off} \wedge (\mathit{on} \supset G)}\ \wedge_R}{\Gamma, \mathit{off} \longrightarrow \mathit{toggle}(G)}\ \supset_L}{\Gamma \longrightarrow \mathit{off} \supset \mathit{toggle}(G)}\ \supset_R$$

So, rather than being toggled, the switch has indeterminate state during the proof of G. The problem is the implicit use, in the appplication of the $\wedge R$ rule, of the contraction rule of intuitionistic logic which allows the original state of the switch to be copied to both sides of the proof tree.

By considering linear management of proof contexts, in which the use of contraction and weakening is restricted to formulas marked with the ! operator, this and several other similar problems can be properly modeled. For instance, if the horn clauses above are replaced with the following linear logic formulas:

$$\Gamma = \left\{ \begin{array}{l} !\{\forall G.[\mathit{toggle}(G) \circ\!- (\mathit{on} \otimes (\mathit{off} \multimap G))]\} \\ !\{\forall G.[\mathit{toggle}(G) \circ\!- (\mathit{off} \otimes (\mathit{on} \multimap G))]\} \end{array} \right.$$

then the proof of the equivalent goal, *off* $\multimap$ *toggle*(G) proceeds as:

$$\dfrac{\dfrac{}{off \longrightarrow off} \quad \dfrac{\dfrac{\vdots}{\Gamma, on \longrightarrow G}}{\Gamma \longrightarrow on \multimap G}\multimap_R}{\dfrac{\dfrac{\Gamma, off \longrightarrow off \otimes (on \multimap G)}{\Gamma, off \longrightarrow toggle(G)}\multimap_L}{\Gamma \longrightarrow off \multimap toggle(G)}\multimap_R}\otimes_R$$

with the desired result that the switch is in the toggled position during the proof of G.

In two recent papers Miller and I have discussed at length the design of a logic programming language based on such formulas [4, 6]. Inference rules for the operators of the language are given in Figure 1. While these rules are not the standard ones of linear logic, they are equivalent to a fragment of linear logic. In this system a proof context consists of two parts: the intuitionistic part (on the left of the semi-colon), in which arbitrary implicit contraction and weakening are allowed, and the linear part (on the right of the semi-colon), in which those rules are barred.

Concrete Syntax and Relationship with λProlog

An important aspect of the Lolli project was the hope that the language could be designed as a modular refinement of λProlog. That is, any purely λProlog program should run 'unmodified' within Lolli[2] and behave in the expected way.

Since the logical operators of the two languages are different, this embedding requires defining a mapping of formulas of intuitionistic logic into the new system. Girard gave such a mapping in the first paper on linear logic [2]. However, given that we are working in the restricted setting of hereditary Harrop formulas it is possible to define a more parsimonious, albeit more complicated, one. This translation, was introduced in a previous paper [6], and is in the form of two mutually recursive functions, one applied to formulas in negative positions (ie. program clauses), and the other to formulas in positive positions (ie. queries).

$$\begin{aligned}
(A)^+ = (A)^- &= A, \text{ where } A \text{ is atomic} \\
(true)^+ = 1 &\qquad (true)^- = \top \\
(B_1 \wedge B_2)^+ &= (B_1)^+ \otimes (B_2)^+ \\
(B_1 \wedge B_2)^- &= (B_1)^- \,\&\, (B_2)^- \\
(B_1 \supset B_2)^+ &= (B_1)^- \Rightarrow (B_2)^+ \\
(B_1 \supset B_2)^- &= (B_1)^+ \multimap (B_2)^- \\
(\forall x.B)^+ &= \forall x.(B)^+ \\
(\forall x.B)^- &= \forall x.(B)^- \\
(B_1 \vee B_2)^+ &= (B_1)^+ \oplus (B_2)^+ \\
(\exists x.B)^+ &= \exists x.(B)^+
\end{aligned}$$

[2]The current implementation of Lolli is an essentially first-order language (ie., while it allows quantification over predicates, formulas, and terms, it does not implement λ-terms or higher-order unification), so this section should be read as referring to the similar fragment of λProlog.

The intuitionistic sequent (over just these operators) $\Gamma \longrightarrow G$ is then mapped to the sequent $\Gamma^-;\emptyset \longrightarrow G^+$, which has a proof if and only if the original sequent did.

Given the λProlog syntax for hereditary Harrop formula programs, this mapping suggests a concrete syntax for the operators of the language, which is given in the following table:

Operator	Parity	Syntax
$\top$	+	`erase`
1	+	`true`
&	+	`&`
	-	`&`
$\otimes$	+	`,`
$\oplus$	+	`;`
$-\!\circ$	+	`-o`
	-	`:-`
$\Rightarrow$	+	`=>`
	-	`<=`
!	+	`{...}`
$\forall x.B$	+	`forall x\B`[3]
	-	`forall x\B*`
$\exists x.B$	+	`exists x\B*`

As with λProlog, terms and atoms are written in a curried form and the standard quantifier assumptions are made. It is straightforward to confirm that existing Prolog and λProlog programs are written, and run, as expected. For instance, the λProlog query:

```
pi X\ pi Y\
   (memb X (X::Y))  =>
pi X\ pi Y\ pi Z\
   (memb X (Y::Z) :- neq X Y, memb X Z) =>
memb G (a::b::nil).
```

represents the formula:

$$\exists G.[(\forall X.\forall Y.memb(X, X :: Y)) \supset \\ (\forall X.\forall Y.\forall Z.(memb(X, Y :: Z) \subset (neq(X, Y) \wedge memb(X, Z)))) \supset \\ memb(G, a :: b :: nil)]$$

which, when translated into the new system using the $()^+$ translation, becomes:

$$\exists G.[(\forall X.\forall Y.memb(X, X :: Y)) \Rightarrow \\ (\forall X.\forall Y.\forall Z.(memb(X, Y :: Z) \circ\!- (neq(X, Y) \otimes memb(X, Z)))) \Rightarrow \\ memb(G, a :: b :: nil)]$$

which has the concrete syntax:

[3]The use of **forall** and **exists** as syntax for the explicit quantifiers represents a personal preference of this author.

```
forall X\ forall Y\
   (memb X (X::Y))  =>
forall X\ forall Y\ forall Z\
   (memb X (Y::Z) :- neq X Y, memb X Z) =>
memb G (a::b::nil).
```

And, when run, this query will have the same execution profile as the original λProlog query.

In contrast, programs which take advantage of the linear features of the system will of necessity make use of the new elements of the syntax. So, for instance, the ill-performing intuitionistic formulas defining the *toggle* predicate would be written (in λProlog and Lolli) as:

```
toggle G :- on, off => G.
toggle G :- off, on => G.
```

while the well-performing linear logic formulas would be written as:

```
toggle G :- on, off -o G.
toggle G :- off, on -o G.
```

In order for existing programs to work properly, it is assumed that the clauses in a module are loaded into the unbounded (intuitionistic) portion of the proof context. The programmer can override this assumption by preceding individual clauses with the `LINEAR` declaration. Thus, it is possible to specify an initial setting for the switch within the program file, as in:

```
LINEAR on.
```

Note that the use of all uppercase for `LINEAR`, is not optional. Since the system uses curried notation, this is the only way (short of ruling out its use in other forms) of recognizing that it is a declaration, and not a predicate name. For consistency, and improved readability, this restriction is also applied to the `LOCAL` and `MODULE` declarations described below.

Modules

Lolli programs are divided into modules in the same way as λProlog programs. By convention, enforced by the interpreter, files carry the extension '`.ll`', and, by analogy to the λProlog `==>` operator, are loaded using the operator '`--o`'. The command '`load modulename`', which is equivalent to '`modulename --o top`', is also available.

A module may begin with a list of local constant declarations, such as:

```
LOCAL a B c.
LOCAL d.
```

with multiple constants separated by spaces, or listed in separate declarations. Because Lolli is essentially first-order, types and kinds, and their declarations, are not needed or supported. A future release of Lolli may support L_λ-unification [7], but will likely still be type-free. Note that

since constants are untyped, predicate names may be reused at different arities, as in ordinary Prolog.

The λProlog module system has been extended to allow for parameterized modules. That is, the module declaration is of the form:

```
MODULE modname param_1 ... param_n.
```

where `modname` matches the root of the file name, and the parameters are variables to be unified placewise with the terms in the loading goal. Note that while the formal parameters are variables, they are generally intended to be viewed as constants within the module, and as such may begin with lowercase characters if the programmer so chooses. Thus, if the module is declared:

```
MODULE foo a B.
```

and is loaded with '`foo c d --o top`', then the clauses in `foo.ll` are loaded with all instances of `a` and `B` instantiated to `c` and `d` respectively.

The logical status of the module system can be summarized as follows, the declaration:

MODULE mod $x_1 \ldots x_n$.
LOCAL $y_1 \ldots y_m$.

$H_1 x_1 \ldots x_n y_1 \ldots y_m z_{1_1} \ldots z_{q_1}$.
$\vdots \quad \vdots$
LINEAR $H_i x_1 \ldots x_n y_1 \ldots y_m z_{1_i} \ldots z_{q_i}$.
$\vdots \quad \vdots$
$H_p x_1 \ldots x_n y_1 \ldots y_m z_{1_p} \ldots z_{q_p}$.

associates to `mod` the parameters $x_1 \ldots x_n$, the local constants $y_1 \ldots y_m$, and the clauses $H_1 \ldots H_p$, which may contain free occurrences of the variables $x_1 \ldots x_n$ and constants $y_1 \ldots y_m$. Each clause H_i may also contain free occurances of the otherwise undeclared variables $z_{1_i} \ldots z_{q_i}$. When the module is loaded within a goal formula, using the syntax `mod` $t_1 \ldots t_n$`--o` B, that goal is considered only as short-hand for the goal

forall y_1\ ...forall y_m\[
  forall z_{1_1}\ ...forall z_{q_1}\($H_1 t_1 \ldots t_n y_1 \ldots y_m z_{1_1} \ldots z_{q_1}$) =>
  $\vdots$
  forall z_{1_i}\ ...forall z_{q_i}\($H_i t_1 \ldots t_n y_1 \ldots y_m z_{1_i} \ldots z_{q_i}$) -o
  $\vdots$
  forall z_{1_p}\ ...forall z_{q_p}\($H_p t_1 \ldots t_n y_1 \ldots y_m z_{1_p} \ldots z_{q_p}$) => B].

Here, we overload the symbols $y_1, \ldots, y_m$ to be constants in the `LOCAL` declaration and bound variables in the displayed formula above. In general. this overloading should not cause problems.

Also, in this example, it is assumed that the formula B and the terms $t_1, \ldots, t_n$ do not contain occurrences of $y_1, \ldots, y_m$. Finally, it is assumed that $y_1 \ldots y_m$, $t_1 \ldots t_n$, $x_1 \ldots x_n$, and $z_{1_1} \ldots z_{q_p}$ are all pairwise disjoint.

The implementation of parameterized modules was driven by the need to be able to handle the object-oriented programming examples from an earlier paper [3], where they were used to pass initialization information to objects. Nevertheless they have proved useful in a number of instances. For example, the following module defines the shell of a multiset rewriting system, along the lines of the example given in [4, 6]. The rewrite rules themselves, however, are in a separate module, whose name is passed to this one as a parameter when this module is loaded. In order to ensure the soundness of the rewriter, a local predicate name is used to store the multiset in the database. That name is, in turn, passed to the rules module when it is loaded. The shell is given by:

```
MODULE rewrite rulemodule.

LOCAL hyp.

collect nil.
collect (X::L) :- hyp X, collect L.

unpack nil G :- G.
unpack (X::L) G :- hyp X -o unpack L G.

rewrite L K :- unpack L ((rulemodule hyp) --o (rewrite  (collect K))).
```

while a rule module might be of the form:

```
MODULE rules1 hyp.

rewrite G :- G.

rewrite G :- hyp 4, ((hyp 2, hyp 2) -o rewrite G).
rewrite G :- hyp 4, ((hyp 3, hyp 1) -o rewrite G).
rewrite G :- hyp 3, ((hyp 2, hyp 1) -o rewrite G).
rewrite G :- hyp 2, ((hyp 1, hyp 1) -o rewrite G).
```

and a sample query would be:

```
?- rewrite rules1 --o rewrite (3::nil) L.

?L674 <- (3 :: nil) .;
?L674 <- (2 :: 1 :: nil) .;
?L674 <- (1 :: 2 :: nil) .  ...
```

Implementation

Lolli is currently available in two implementations. The first is a simple Prolog meta-interpreter given in [4, 6] and reproduced in Figure 2. The code as given implements only the propositional fragment of the language (with a few differences from the concrete syntax described above), but is useful for experimenting with the core of the underlying logic. The meta-interpreter could be trivially extended to the first-order language by re-implementing it in λProlog. Other than the change of syntax, that system would differ only in the addition of two clauses to handle quantification. Unfortunately, the lack of op declarations in λProlog would make the system a little more unwieldy.

The author has also developed a relatively rich implementation of Lolli in Standard ML of New Jersey (which should port to any ML which can handle MLYACC and MLLEX). That implementation supports the full language as described here, in addition to a reasonable selection of evaluable predicates and one extra-logical control structure (guard expressions). That implementation was inspired by (and built on a core of code from) Elliott and Pfenning's article on implementing λProlog-like languages in a functional setting [1]. The full implementation of Lolli, with documentation, many example programs, and DVI files for several relevant papers, is available by anonymous ftp from `ftp.cis.upenn.edu` (`130.91.6.8`) in the directory `/pub/Lolli`. If you retrieve the system, please send mail to `hodas@saul.cis.upenn.edu` so that you may be informed of updates.

Conclusion

The Lolli project is an ongoing one, and the language is by no means frozen. On the other hand, the collection of program examples is growing [4, 6, 5], and this shows that the logic fragment chosen represents a useful extension of the traditional hereditary Harrop formulas of λProlog.

Acknowledgements

The author is grateful to Dale Miller, for his partnership in this work, and to Jean-Marc Andreoli, Gianluigi Bellin, Jawahar Chirimar, Remo Pareschi, Pat Lincoln, Andre Scedrov, James Harland, Jean-Yves Girard and Fernando Pereira for conversations (with the author and with Dale Miller) about aspects of the design and theory of Lolli. He is also grateful to Frank Pfenning and Conal Elliott for providing such a strong base to work with in implementation. Finally, to Elizabeth Hodas for helpful editorial comments.

```
% The logic being interpreted contains the following logical connectives:
%   true/0            a constant (empty tensor, written as 1 in the logic)
%   erase/0           a constant (erasure, written as Top in the logic)
%   bang/1             the modal, written as {} in the paper.

:- op(145,xfy,->).  %   linear implication, written as -o in the paper
:- op(145,xfy,=>).  %   intuitionistic implication
:- op(140,xfy,x ).  %   multiplicative conjunction (tensor)
:- op(140,xfy,& ).  %   additive conjunction
:- op(150,xfy,::).  %   non-empty list constructor

   interp(G) :- prove(nil, nil, G).

   isG(true).                              isR(erase).
   isG(erase).                             isR(B)        :- isA(B).
   isG(B)        :- isA(B).                isR(B1 & B2)  :- isR(B1), isR(B2).
   isG(B1 -> B2) :- isR(B1), isG(B2).      isR(B1 -> B2) :- isG(B1), isR(B2).
   isG(B1 => B2) :- isR(B1), isG(B2).      isR(B1 => B2) :- isG(B1), isR(B2).
   isG(B1 &  B2) :- isG(B1), isG(B2).
   isG(B1 x  B2) :- isG(B1), isG(B2).
   isG(bang(B))  :- isG(B).

   prove(I,I, true).
   prove(I,O, erase)   :- subcontext(O,I).
   prove(I,O, G1 & G2) :- prove(I,O,G1), prove(I,O,G2).
   prove(I,O, R -> G)  :- prove(R :: I, del :: O,G).
   prove(I,O, R => G)  :- prove(bang(R) :: I, bang(R) :: O,G).
   prove(I,O, G1 x G2) :- prove(I,M,G1), prove(M,O,G2).
   prove(I,I, bang(G)) :- prove(I,I,G).
   prove(I,O, A)       :- isA(A), pickR(I,M,R), bc(M,O,A,R).

   bc(I,I,A, A).
   bc(I,O,A, G -> R)  :- bc(I,M,A,R), prove(M,O,G).
   bc(I,O,A, G => R)  :- bc(I,O,A,R), prove(O,O,G).
   bc(I,O,A, R1 & R2) :- bc(I,O,A,R1); bc(I,O,A,R2).

   pickR(bang(R)::I, bang(R)::I, R).                subcontext(del::O, R ::I) :- isR(R), subcontext(O,I).
   pickR(R::I,       del::I,     R) :- isR(R).      subcontext(S::O,   S::I)  :- subcontext(O,I).
   pickR(S::I,       S::O,       R) :- pickR(I,O,R).  subcontext(nil,    nil).

% The following code provides the hooks into application programs.
:- op(150,yfx,<-). %     the converse of the linear implication

% Applications using this interpreter are specified using the <-/2 functor (denoting the converse
% of linear implication). We shall assume that clauses so specified are implicitly banged (belong
% to the unbounded part of the initial context) and that the first argument to -> is atomic.  The
% following clause is the hook to clauses specified using <-.

prove(I,O, A) :- isA(A), A <- G, prove(I,O,G).

% A few input/output non-logicals.

prove(I,I, write(X)) :- write(X).        prove(I,I, read(X)) :- read(X).        prove(I,I, nl) :- nl.

% The following is a flexible specification of isA/1
notA(write(_)).  notA(read(_)).  notA(nl).  notA(erase).  notA(true).  notA(del).
notA(_ & _).  notA(_ x _).  notA(_ -> _).  notA(_ => _).  notA(bang(_)).
isA(A) :- \+(notA(A)).
```

Figure 2: A Prolog implementation of Lolli

References

[1] Conal Elliott and Frank Pfenning. A semi-functional implementation of a higher-order logic programming language. In Peter Lee, editor, *Topics in Advanced Language Implementation*, pages 289 – 325. MIT Press, 1991.

[2] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[3] Joshua Hodas and Dale Miller. Representing objects in a logic programming language with scoping constructs. In David H. D. Warren and Peter Szeredi, editors, *1990 International Conference in Logic Programming*, pages 511 – 526. MIT Press, June 1990.

[4] Joshua Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic: Extended abstract. In G. Kahn, editor, *Sixth Annual Symposium on Logic in Computer Science*, pages 32 – 42, Amsterdam, July 1991.

[5] Joshua S. Hodas. Specifying filler-gap dependency parsers in a linear-logic programming language. In Krystopf Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming, Washington D.C.*, 1992.

[6] Joshua S. Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Journal of Information and Computation*, 1992. To appear.

[7] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497 – 536, 1991.

Some Kind of Magic for (a Restriction of) L_λ

Alain Hui Bon Hoa [1]
INRIA Rocquencourt–Ecole Normale Supérieure
huibonho@margaux.inria.fr

1 Abstract

Higher-Order Hereditary Harrop (HOHH) formulas have been seriously studied in the latest years as a basis for higher-order logic programming languages, resulting in several implementations. Yet, no alternative to SLD-resolution has been developed for these languages, while for instance some Bottom-Up strategy would allow extensions of Prolog applications in such domains as language analysis, deductive databases or software engineering.
We studied a restriction of the higher-order language L_λ, which we named l_λ, and for which we could define a sound and complete Bottom-Up resolution strategy. This strategy turns out to be very simple, the unification taking care of all the constraints due to quantification over function variables. We believe that this is a first step towards fully exploiting higher-order logic in several application fields. As an example, we study the use of the Magic Set method, developed in the database community and which, when applied to Horn Clauses, solves the problem occurring in a naïve Bottom-Up resolution of computing a great deal of useless facts. We present here an extension of the Magic Set method to our higher-order language l_λ.

2 Introduction

The perspective of higher-order logic programming languages has been deeply studied lately. Their interest as meta-programming languages and more perspicuous formalisms has been established and argued by many authors. It was proved that Higher-Order Hereditary Harrop (HOHH) formulas formed a good basis for such a language, since its higher-order features still accept uniform proofs [12] and thus support a proof strategy extending SLD-resolution (used in Prolog).

As a result, the language λ-Prolog was developed [13] implementing HOHH formulas. Gérard Huet's results on higher-order unification made it possible to handle the unification involved in the proving procedure. Several implementations of λ-Prolog have been given such as Prolog/Mali, eLF, ...The problems due to undecidability and possible lack of a most general unifier in higher-order unification have been eliminated in L_λ, a restriction of λ-Prolog, with an acceptable loss of expressive power. Being a logic programming language with λ-abstraction, function variables, and simple unification [10], L_λ both presents higher-order nice features and is likely to support efficient implementations.

Yet it still suffers from the lack of an alternative to the extension of SLD-resolution originally presented: to our knowledge, no real attempt to a Bottom-Up resolution strategy has been con-

[1] This work was partially supported by a grant from a European Software Factory (ESF) project.

sidered. In such domains as language analysis [21], deductive databases [2] or software engineering [18], Horn Clauses are efficiently used with some kind of Bottom-Up strategy which is complete and more suitable to the concerned application. The extension of such a strategy to L_λ would allow to apply its higher-order logic in these fields, resulting in appropriate implementations of methods already studied for λ-Prolog by Eugene Rollins and Jeannette Wing [20] as well as François Rouaix [19] for search in libraries, and Dale Miller and Gopalan Nadathur in language analysis [11].

As a first step to this aim, we have studied a fragment of L_λ, which we called l_λ. We discuss the choice of this fragment of HOHH at the end of the paper. This logic programming language supports λ-abstraction, function variables and quantification, but does not authorize implication in clause bodies (which L_λ does). The main reason for this restriction is that dealing with implication in clause bodies requires some process introducing and discharging assumptions, which is quite difficult to achieve in a Bottom-Up resolution. As a restriction of L_λ, l_λ also makes use of its simple and decidable unification algorithm.
We proved that a sound and complete Bottom-Up strategy was possible for L_λ, resulting in a simple interpreter [7]. While SLD-resolution requires the use of *quantifier prefixes* to encode the different constraints over quantified function variables, our interpreter notably presents the advantage not to need any *quantifier prefix.*

This Bottom-Up interpreter represents a first step towards offering higher-order logic features to fields for which SLD is not the most adequate evaluation procedure. For instance, in database systems, it is for computational reasons advantageous to consider set-oriented query-processing procedures. But naïve Bottom-Up strategies tend to do a great deal of unnecessary work. A nice solution developed first in the database community [4] and then extended to logic programming and the Horn Clauses formalism [17] is the method of Magic Sets. This method transforms a logic program $\mathcal{P}$ and a goal into another program $Magic\ (\mathcal{P})$ which, when evaluated Bottom-Up, mimics the SLD-resolution of $\mathcal{P}$. This method therefore solves the problem of useless computations in a Bottom-Up strategy. We have studied an extension of the Magic Set method to our language l_λ which could present the same advantages. Unfortunately, the direct extension of Magic Sets to l_λ is not possible since it leads to transformed programs which are no more in l_λ (nor even in L_λ). We propose a sound and complete method relying on the basic principles of the Magic Set method, and which, when evaluating the transformed program Bottom-Up, mimics a SLD-strategy prediction and performs a Bottom-Up resolution from the relevant axioms. This method computes more facts than an exact SLD-resolution, but significantly restricts the space search of a naïve Bottom-Up evaluation.

This paper is organized as follows: in section 2. we briefly describe the language l_λ and we outline the interpreter originally developed for L_λ in [10] and extending first-order SLD-resolution. We then sketch in section 3 how l_λ can support a simple Bottom-Up interpreter. More details on this part may be found in [7]. In section 4, we present the main result of this paper, namely some extension of the Magic Set method to our higher-order language l_λ. Introducing a partial notion of subsumption between prefixed terms, we show how this method rewrites a program into another which, when evaluated Bottom-Up, mimics a Top-Down prediction mixed with a Bottom-

Up resolution. We prove this method sound and complete. We finally discuss some possible extensions to our language.

3 The higher-order logic programming language l_λ

We present here the constraints induced by the higher-order logic nature of the variables, and the logic programs used in l_λ. We sketch the interpreter deduced from Dale Miller's one for L_λ. The reader familiar with L_λ may skip this section.

Clauses used in l_λ are the usual Horn clauses extended with function variables, λ-expressions and universal quantification. A condition on the syntax of terms ensures the decidability and the existence of a most general unifier (m.g.u.) in case of success of the unification algorithm. This allows to apply a proof method which can be conceived as an extension of the SLD-resolution used in Prolog (and will sometimes also be referred as SLD in the rest of the paper). Complete details may be found in [10].

3.1 Extended Horn Clauses

In L_λ, terms are simply-typed λ-terms. As it is of no great incidence in our purpose and for the sake of simplicity, we will consider here untyped λ-terms. We deal in l_λ with Horn Clauses extended in three ways:

- λ-expressions, which means we can use λ-abstraction to represent functions and that the interpreter is able to synthesize λ-functions.

- function variables, which means we can use free variables to represent functions, and have them instantiated either by functions originally defined in the program or by synthesized λ-functions.

- universal quantifications, which have different interpretations according to where in the clauses they are used:

  Consider for instance the following logic program:

  $$Query \leftarrow \forall x\ P(f(x))$$
  $$\forall y\ (P(y) \leftarrow Q(y))$$

  In this example, the quantified variable x is placed on the right of $\leftarrow$. It is then said to be *essentially universal*: the goal requires we prove $P(f(x))$ for all x. The usual treatment of this case is to replace x by an *eigen-variable*, i.e. a new constant. This essentially universal variable therefore cannot be instantiated. For this reason, variables bound by a λ-abstraction, which cannot be instantiated either, will also be said essentially universal.

The quantified variable y situated on the left of $\leftarrow$ is said to be *essentially existential*: on the contrary of x, it may be instantiated in order to prove a goal, in this case by $f(x)$.

The authorized definite clauses therefore have the following form:

$$\forall \vec{x}\ (\forall \vec{x_A}\ A\ \leftarrow\ \forall \vec{x_1}\ B_1\ \wedge \forall \vec{x_2}\ (\forall \vec{x_{21}} B_{21} \ldots))$$

where

- $\vec{x}$ represent a set of essentially existential variables which may appear in any term of the quantified formulae A, B_1, ...

- $\vec{x_A}$ are also essentially existential variables, but which appear only in A.

- the $\vec{x_i}$ are essentially universal variables which may appear in each of the formulae quantified by $\forall \vec{x_i}$.

Moreover a condition denoted by (#) is set on the form of the terms: in any application $(xt_1t_2 \ldots t_n)$ where x is an essentially existential variable, the t_i's are required to be distinct essentially universal variables, quantified on the right (in the scope) of x. This guarantees the decidability of the unification and the existence of a (in some sense) unique m.g.u. [10].

The following example using these extensions gives an idea of the problems they raise:

Example 1 *Consider the following program:*

$$\begin{array}{rl} \forall x \forall l \forall k \forall m & (append\ (cons\ x\ l)\ k\ (cons\ x\ m)\ \leftarrow\ append\ l\ k\ m) \\ \forall k & (append\ nil\ k\ k) \end{array}$$

and the goal formula Query $\leftarrow\ \forall y\ (append\ (cons\ a\ nil)\ y\ Z)$.
Notice that the unknown Z *is implicitly quantified by* $\exists Z\ \forall y$, *which means* Z *cannot depend on* y.

Then an SLD-like resolution would roughly proceed this way:

We replace the essentially universal variable y *by an eigen-variable* $\tilde{y}$ *and prove the goal*

$$(append\ (cons\ a\ nil)\ \tilde{y}\ Z)$$

We unify this formula with the head of the first clause, obtaining the substitution

$$\left\{ \begin{array}{rl} x \longmapsto & a \\ l \longmapsto & nil \\ k \longmapsto & \tilde{y} \\ Z \longmapsto & cons\ a\ m \end{array} \right.$$

and the new goal: $(append\ nil\ \tilde{y}\ (cons\ a\ m))$.

Unifying this new goal with the head of the second clause, we get the final substitution

$$\begin{cases} x \mapsto & a \\ l \mapsto & nil \\ k \mapsto & \tilde{y} \\ m \mapsto & \tilde{y} \\ Z \mapsto & cons\ a\ \tilde{y} \end{cases}$$

This would lead to a solution for the initial goal (contrary to the intuition), if we hadn't first specified that Z could not be instantiated by a term containing y. Therefore the resolution leads to no solution, which was the correct and expected answer.

If we had considered instead the goal formula $Query \leftarrow \forall y\ (append\ (cons\ a\ nil)\ y\ (H y))$, *where H is a function variable, the same resolution would have lead to two solutions:*

$$\begin{array}{rl} H \mapsto & \lambda u \bullet cons\ a\ \tilde{y} \\ and \quad H \mapsto & \lambda u \bullet cons\ a\ u \end{array}$$

Again, the condition on H eliminates the first solution and we get the expected higher-order answer:

$$H \mapsto \lambda u \bullet cons\ a\ u$$

This example shows that a correct resolution of our programs needs to retain

- which variables are essentially existential and which are essentially universal.
- their order of appearance during the resolution, so that we can define for each essentially existential variable the appropriate essentially universal variables its substitution terms may contain.

This will be done in the SLD interpreter by quantifier prefixes, and in the following, we will underline these prefixes of quantifiers indicating whether a variable is essentially existential or universal, to tell them from the syntactic symbols in the formulae.

3.2 An SLD interpreter

We formalize here the method used in the previous example. The interpreter is a restriction of Dale Miller's one for L_λ: the strategy extends SLD-resolution, the constraints over the variables are encoded in quantifier prefixes. The unification algorithm is sketched on an example.

To present the interpreter, we introduce a simple meta-logic containing the logical constants $\wedge$, $\top$ (true), $\bot$ (false), $\underline{\forall}$, and $\underline{\exists}$. The atomic propositions of this meta-logic are then either the constants $\top$ or $\bot$, or a sequent judgement $\mathcal{P} \Rightarrow G$, or an equality judgement $t = s$. The sequent judgement intuitively corresponds to the notion of goal to prove and the equality judgement to that of unification.
The interpreter deals with closed quantified formulas of the meta-logic, the constraints over the variables being encoded by these meta-level quantifications:

for instance, $\underline{Q} = \underline{\exists x_1 \forall x_2 \forall x_3 \exists x_4}$ retains that x_1 and x_4 are essentially existential variables while x_2 and x_3 are essentially universal ones; moreover x_4 is in the scope of x_2 and x_3 (i.e. its substitution terms may contain x_2 and x_3) while x_1 is not.

The resolution is initialized with $\underline{Q_0}$ $(\mathcal{P} \Rightarrow G)$ where $\underline{Q_0}$ contains the constants of the program $\mathcal{P}$ followed by the unknown variables in G, the initial goal. It ends when there is only a logical constant left, $\top$ meaning a success and $\bot$ a failure.

The interpreter then appears as rules over this meta-logic:

AND A sequent of the form $\mathcal{P} \Rightarrow G_1 \wedge G_2$ is replaced with the conjunction of sequents

$$(\mathcal{P} \Rightarrow G_1) \ \wedge \ (\mathcal{P} \Rightarrow G_2)$$

.

GENERIC A sequent of the form $(\mathcal{P} \Rightarrow \forall x\ G)$ is replaced with the sequent

$$\underline{\forall \tilde{x}}\ (\mathcal{P} \Rightarrow [x \leftarrow \tilde{x}]G)$$

where $\tilde{x}$ is a new symbol.

BACKCHAIN A sequent of the form $\mathcal{P} \Rightarrow A$ is replaced with the sequent

$$\underline{\exists \vec{x} \exists \vec{x}_B}\ ((A = B) \ \wedge \ (\mathcal{P} \Rightarrow D))$$

if the program contains a clause $\forall \vec{x}(\forall \vec{x}_B B \ \leftarrow \ D)$

If no such clause exists, then we have a failure in the search branch, which we represent by replacing the above sequent by the constant $\bot$.

Quantified equality judgements are treated by unification. Provided the correctness of the unification algorithm, this interpreter can be proved sound and complete [10].
In the following we will keep the same notation x, even when it should be replaced by the eigenvariable $\tilde{x}$.

3.3 Unification in l_λ

Though general higher-order unification was proved undecidable [6], in the case of L_λ and thus of l_λ, the condition (#) required on the terms leads to a correct and decidable algorithm [9], which provides us with a m.g.u. in case of success of the unification. For some reasons which will become clearer in section 4, and because this restores a symmetry in the presentation, we prefer to view this unification as one between two prefixed terms $\underline{Q_A}$ A and $\underline{\exists \vec{x_B}}$ B, while it is originally and usually presented as an unification of the two terms A and B under the mixed prefix $\underline{Q_A \exists \vec{x}_B}$. Thus $UNIFY(\underline{Q_A}\ A, \underline{\exists \vec{x_B}}\ B)$ will be computed using the traditional algorithm denoted by $Unify(\underline{Q_A \exists \vec{x}_B},\ A = B)$.

This view also presents the advantage to separate clearly the resolution part from the unification one: the algorithm then does not consist in appending the quantifiers of a head of a clause $\underline{\exists \vec{x}_B}$ to that of a goal $\underline{Q_A}$, but to encode the scope constraints of each variable, thus to reveal the dependencies between the most "flexible" ones $\underline{\exists \vec{x}_B}$ and the essentially universal ones in $\underline{Q_A}$.
We sketch this algorithm on an example:

Example 2 *Consider the problem UNIFY* $(\underline{\forall m \exists x \forall y \forall z \forall w}\ f(xy)z\ , \underline{\exists u \exists v}\ f(\lambda a \bullet u)v)$. *The algorithm proceeds this way:*

we first write it under the form

$$\underline{\forall m \exists x \forall y \forall z \forall w \exists u \exists v} \qquad f(xy)z = f(\lambda a \bullet u)v$$

as the functional symbols are essentially universal and identical, we compare their arguments

$$\underline{\forall m \exists x \forall y \forall z \forall w \exists u \exists v} \qquad \begin{cases} xy = \lambda a \bullet u \\ z = v \end{cases}$$

λ- abstraction is treated by using the extensionality property $M = \lambda x \bullet Mx$

$$\underline{\forall m \exists x \forall y \forall z \forall w \exists u \exists v \forall a} \qquad \begin{cases} xya = u \\ v = z \end{cases}$$

we then reveal the dependencies of $\underline{\exists u}$ over essentially universal variables by **raising** *it up to x*

$$\underline{\forall m \exists x \exists \mathbf{u'} \forall y \forall z \forall w \exists v \forall a} \qquad \begin{cases} xya = u'yzw \\ v = z \end{cases} \qquad u \longmapsto u'yzw$$

the irrelevant argument variables are then suppressed by **pruning** *over z and w*

$$\underline{\forall m \exists x \exists \mathbf{u''} \forall y \forall z \forall w \exists v \forall a} \qquad \begin{cases} xya = u''y \\ v = z \end{cases} \qquad \begin{cases} u \longmapsto u'yzw \\ u' \longmapsto \lambda yzw \bullet u''y \end{cases}$$

we finally get the substitution σ :

$$\underline{\forall m \exists u'' \forall y \forall z \forall w \forall a} \qquad \begin{cases} u \longmapsto u''y \\ x \longmapsto \lambda ya \bullet u''y \\ v \longmapsto z \end{cases}$$

This solution is a m.g.u. in the sense that any closed unifier is an instance of σ respecting the constraints encoded by $\underline{\forall m \exists u'' \forall y \forall z \forall w \forall a}$

We have so far obtained a higher-order programming language with an interpreter using an SLD-resolution similar to that used in PROLOG. The constraints between variables are captured by quantifier prefixes which memorize which variables are essentially existential and which are essentially universal, and order them so that a variable is contained in the scope of the variables on its left in the prefix.

4 A Bottom-Up interpreter for l_λ

4.1 Motivations and intuitions

A first motivation for a Bottom-Up interpreter is theoretical: the good properties of the HOHH with respects to uniform proofs allowed the design of a goal directed strategy for proofs, resulting in the SLD method. It is thus interesting to draw the parallel between the languages that stemmed from HOHH and Horn Clauses as far as possible, notably concerning the availability of alternative resolution strategies.

Moreover a Bottom-Up interpreter can be useful in a variety of domains, as can be seen with Horn Clauses applications: in natural language parsing, whose formalism was proved very close to that of logic programming [15], people usually start from the token chain to be analyzed and deduce its structure (Bottom-Up approach) rather than compute a possible structure and try it on the chain (Top-Down approach). In deductive databases, a Bottom-Up strategy, close to the least fixed point semantics, makes use of set-oriented query-answering procedures, which are more efficient ways of processing queries in this field [1, 2]. Moreover it presents the important property of being operationally complete.

We have therefore been interested in studying a Bottom-Up interpreter for l_λ. Although the principle is quite simple, relying on the modus ponens rule

$$\frac{(p \leftarrow q \wedge r) \quad r}{(p \leftarrow q)}$$

its application to our higher-order language is a little tricky for two main reasons:

1. The modus ponens schemata applies to a conjunction of atoms (i.e. , of particular formulas we are able to unify). l_λ bodies of clauses contain nested quantifications and conjunctions, so we may have to deal with conjunctions of arbitrarily complex formulas.

2. The quantification of functional variables involves possible constraints over them: this problem is addressed by quantifier prefixes in SLD, but has to be considered specifically in a Bottom-Up strategy.

We designed a Bottom-Up interpreter [7], which turns out to be as simple as the one for Horn Clauses. The way we solved the problems described above relies on an analogy between l_λ quantified atoms and Horn Clause atoms. We exploit this analogy to extend the Bottom-Up resolution for Horn Clauses to l_λ. An intuition of this extension is given here; more precisions will be given in the rest of the chapter, and a complete justification may be found in [7]:

Universal quantifiers may be distributed over conjunctions in bodies of clauses, and we obtain logically equivalent formulas, according to the tautology:

$$\forall x\ (A \wedge B) \Leftrightarrow (\forall x\ A) \wedge (\forall x\ B)$$

By applying inductively this transformation, we obtain programs of clauses whose bodies are conjunctions of quantified atoms. We may then formally apply the modus ponens schemata, using higher-order unification of prefixed terms (as we presented it in 2.3). The prefix for a quantified atomic goal may be easily computed by appending the list of the universally quantified variables to the list of its (essentially) existential variables (remember this prefix is only an encoding of the variables present in the term). This schemata may be represented by the following rule:

$$\frac{\forall \vec{x}(\vec{x}_A A \ \leftarrow \ \forall \vec{y}\ B) \quad \forall \vec{u}\ C}{\forall \vec{x}' \sigma(A)} \text{ if } UNIFY\ (\underline{\exists \vec{x} \forall \vec{y}}\ B \ = \ \underline{\exists \vec{u}}\ C) \ = \ (Q, \sigma)$$

where $\vec{x}'$ contains the (essentially existential) variables in $\sigma(A)$

This formal mechanism will be proved correct and looks very simple. In particular, one may notice that no quantifier prefix needs to be kept during the resolution; it is synthesized at each unification step. The reason is that, in a Top-Down resolution, constraints over variables have to be dynamically accumulated and propagated along the search tree, as bindings are. In a Bottom-up strategy, on the other hand, we reason from facts and derive other facts which we may then re-use without knowing their origins. As axioms and heads of clauses only contain essentially existential variables, no constraint is set on them and thus no constraint need be propagated during the computation. Quantifier prefixes are only needed in the unification step, to specify the scope constraints on the variables in the terms to unify. As these constraints are local, they may be statically computed.

The following example gives an intuition of the Bottom-Up procedure applied to the same program as in example 1:

Example 3 *Let $\mathcal{P}$ be the program*

$$\begin{array}{ll} \forall x \forall l \forall k \forall m & (append\ (cons\ x\ l)\ k\ (cons\ x\ m) \ \leftarrow \ append\ l\ k\ m) \\ \forall k & (append\ nil\ k\ k) \end{array}$$

and G the goal $Query \ \leftarrow \ \forall y\ (append\ (cons\ a\ nil)\ y\ (H y))$.

A Bottom-Up resolution would proceed this way:

Starting from the axiom, we chain it with the other clause, obtaining the new axiom:

$$\forall x \forall k\ (append\ (cons\ x\ nil)\ k\ (cons\ x\ k))$$

We chain the new axiom with the desired goal by the unification:

$$UNIFY\ (\underline{\exists H \forall y}\ append\ (cons\ a\ nil)\ y\ (H y) \ = \ \underline{\exists x \exists k}\ append\ (cons\ x\ nil)\ k\ (cons\ x\ k))$$

which identifies x to a and k to y (which is correct as to the scope constraint) and gives:

$$H \longmapsto \lambda u \bullet cons\ a\ u$$

4.2 The Bottom-Up interpreter

The simplicity of this interpreter relies on this remark:
Consider the particular case of a clause of the following form, where A and B are atomic:

$$\forall \vec{x}\ (\forall \vec{x}_A\ A\ \leftarrow\ \forall \vec{y} B)$$

where

$\vec{x}$ represent a set of essentially existential variables appearing either in B or in both A and B.
$\vec{x}_A$ are also essentially existential variables, but which appear only in A.
$\vec{y}$ are essentially universal variables appearing only in B.

Now if we consider a chaining step with the axiom $\underline{\exists \vec{u}}\ C$, we have to realize the unification

$$UNIFY\ (\underline{\exists \vec{x} \exists \vec{x}_A \forall \vec{y}}\ B\ =\ \underline{\exists \vec{u}}\ C)$$

which is computed by

$$Unify\ (\underline{\exists \vec{x} \exists \vec{x}_A \forall \vec{y} \exists \vec{u}}\ .\ C = B)$$

The following remarks hold:

- the $\vec{x}_A$ are left unchanged since they do not appear in the terms to unify. In fact they may even be completely removed from the unification, which we will do hereafter.
- the $\vec{x}$ are not in the scope of the essentially universal $\vec{y}$. Therefore they cannot be substituted by terms containing variables in the scope of the $\vec{y}$. The only essentially existential variables appearing in those substitution terms are then some x or some u' which was raised from a u (and therefore not in the scope of an essentially universal y).
- a variable from $\vec{u}$ may be substituted by terms containing some of the essentially universal $\vec{y}$, but then it cannot appear in any substitution term of one of the $\vec{x}$.

The resulting substitution σ then does not affect the $\vec{x}_A$ and can instantiate the $\vec{x}$ only with terms containing no essentially universal variables $\vec{y}$ or any variable under the scope of a y. As a consequence, $\sigma(A)$ does only contain essentially existential variables, under the scope of no y.

Omitting the quantifiers corresponding to essentially existential variables, as is usually done in λ-Prolog, we thus obtain the following rule presented as a sequent:

$$\frac{(A\ \leftarrow\ \forall \vec{y}\ B)\quad C}{\sigma(A)}\ \text{if } UNIFY\ (\underline{\exists \vec{x} \forall \vec{y}}\ B\ ,\underline{\exists \vec{u}}\ C)\ =\ (Q,\sigma)$$

We therefore obtain a calculus principle very near to that of the first order case, except that the unification is higher-order.

This result can easily be generalized to all kinds of clauses of l_λ, including nested use of quantification and conjunction, on the basis of the following logic equivalences:

$$\begin{array}{rcl} ((B \wedge C) \supset A) & \equiv & (C \supset (B \supset A)) \\ ((\forall x \; (B \wedge C)) \supset A) & \equiv & ((\forall x \; B \wedge \forall x \; C) \supset A) \end{array}$$

More details may be found in [7], justifying the following forward chaining procedure:

Clauses of the general form

$$\forall \vec{x} \; (\forall \vec{x}_A \; A \; \leftarrow \; \forall \vec{y} \; (\forall \vec{y}_1 \; D_1 \; \wedge \; \forall \vec{y}_2 \; D_2))$$

are first transformed into the equivalent ones

$$\forall \vec{x} \; (\forall \vec{x}_A \; A \; - \; \forall \vec{y} \forall \vec{y}_1 \; D_1 \; \wedge \; \forall \vec{y} \forall \vec{y}_2 \; D_2)$$

to which we apply the following rule:

$$\frac{(A \; \leftarrow \; \forall \vec{y} \forall \vec{y}_1 \; D_1 \; \wedge \; \forall \vec{y} \forall \vec{y}_2 \; D_2) \quad B}{\sigma(A) \; \leftarrow \; \forall \vec{y} \forall \vec{y}_1 \; \sigma(D_1)} \; \textbf{if } Unify \; (\underline{\exists \vec{x}_{D_2} \forall \vec{y} \forall \vec{y}_2} \; D_2 \; = \; \underline{\exists \vec{u}_B} \; B) \; = \; (Q, \sigma)$$

This Bottom-Up procedure can be easily proved sound and complete using the deduction rules.

We thus obtain a very simple Bottom-Up interpreter which is very close to the one defined for first order Horn Clauses. Miraculously, all the higher-order features are handled by the higher-order unification which, in the case of l_λ, presents no problem of termination or uniqueness of the m.g.u.

As the rest of the paper is devoted to an application of the Bottom-Up strategy, we will hereafter assume that the clauses of the l_λ programs are written in their expanded form (i.e. bodies of the clauses are conjunction of universally quantified atomic goals).

5 An application: Higher-Order Magic Sets

5.1 First-order Magic Sets

In some fields like deductive databases, computational reasons make it more advantageous to consider forward chaining strategies. Unfortunately, a straightforward Bottom-Up resolution tends to compute many facts useless for the goal to prove. We show this on an example where, unlike usual database conventions, we do not separate intensional and extensional parts.

Example 4 *Consider the following Horn Clauses program:*
$path(X,Y) \; \leftarrow \; edge(X,Y)$
$path(X,Y) \; \leftarrow \; edge(X,Z), \; path(Z,Y)$

$edge(a, b).$
$edge(b, c).$
$edge(d, e).$
$edge(e, f).$
and let the query be
$Query \leftarrow path(a, Y).$
The Bottom-Up processing of this query will compute the complete edge relation and then select the appropriate instances, i.e. all the paths which may be related will be computed, while only those starting from a were required.

On the other hand, SLD resolution presents the advantage of reducing the space of search since the procedure is goal-directed.
To solve this problem, a nice solution was supplied first for databases [4] and then for general Horn clauses [17] by C. Beeri and R. Ramakrishnan, consisting in rewriting a program $\mathcal{P}$ and a query G into a program which, when computed Bottom-Up, mimics a Top-Down evaluation. The rewriting is performed as indicated below:

First-Order Magic Set transformation:
Let $\mathcal{P}$ be a l_λ logic program, G be a goal.
Then $Magic(\mathcal{P})$ is the program obtained by:

- $(magic_G.) \in Magic(\mathcal{P})$
- if $(D \leftarrow G_1 \dots G_n) \in \mathcal{P}$, then $(D \leftarrow magic_D,\ G_1 \dots G_n) \in Magic(\mathcal{P})$
- if $(D \leftarrow G_1 \dots G_n) \in \mathcal{P}$, then $(magic_G_i \leftarrow magic_D,\ G_1 \dots G_{i-1}) \in Magic(\mathcal{P})$ for each $1 \leq i \leq n$

An intuition of the isomorphism between applying an SLD-resolution and evaluating the Magic program Bottom-Up may be found in [14].

Example 5 *This Magic Set transformation produces the following program from the one above, introducing the new predicates magic_path and magic_edge:*
$path(X, Y) \leftarrow magic_path(X, Y),\ edge(X, Y)$
$path(X, Y) \leftarrow magic_path(X, Y),\ edge(X, Z),\ path(Z, Y)$
$edge(a, b) \leftarrow magic_edge(a, b)$
$edge(b, c) \leftarrow magic_edge(b, c)$
$edge(d, e) \leftarrow magic_edge(d, e)$
$edge(e, f) \leftarrow magic_edge(e, f)$
$magic_edge(X, Y) \leftarrow magic_path(X, Y)$
$magic_edge(X, Z) \leftarrow magic_path(X, Y)$
$magic_path(Z, Y) \leftarrow magic_edge(X, Y).\ edge(X, Z)$
$magic_path(a, Z).$
where the prefix magic could be intuitively read as "call".

The computed facts are then the following ones, where the solutions are framed:

$magic_edge(a,Z)$	$magic_edge(a,Y)$	
$edge(a,b)$		
$\boxed{path(a,b)}$	$magic_path(b,Z)$	
$magic_edge(b,Z)$	$magic_edge(b,U)$	
$edge(b,c)$		
$path(b,c)$	$magic_path(c,Z)$	
$\boxed{path\ (a,c)}$	$magic_edge(c,Z)$	$magic_edge(c,V)$

This time, the irrelevant paths concerning the points e, f and g are not computed.

Thus this transformation solves the problem of restricting the set of facts computed during a Bottom-Up resolution.

We study here an extension of this method to our language l_λ, using the Bottom-Up resolution presented in the precedent section.

5.2 Impossibility of a direct extension

A first natural attempt consists in a direct extension of the first-order Magic Set method to l_λ. This leads to a failure, because quantification prevents from rewriting into correct l_λ clauses.

Example 6 *Consider the simple program:*

$B(a,Y)$.

$A(Z) \leftarrow \forall x\ B(Z,p(x))$.

Then a direct application of the Magic Set rewriting would give the following program, with the new predicates magic_A and magic_B:

$B(a,Y) \leftarrow magic_B(a,Y)$

$A(Z) \leftarrow magic_A(Z),\ \forall x\ B(Z,p(x))$

$\ldots \leftarrow magic_A(Z)$

The trouble with the second clause is that we do not know how to transform a quantified goal: knowing that the desired term in the head of the clause should intuitively mean "try to prove $B(Z,p(x))$ for all x", we have to cope with the following problems:

- On one hand, the universal quantification cannot be put out of the magic term (something like $\forall x\ magic_B(Z,p(x))$), since such a quantification in a head of a clause would mean that x is essentially existential, while we want it to be essentially universal.

- On the other hand, to encode that we have to consider the goal $\forall x\ B(Z,p(x))$ in its whole (with x being essentially universal), we can try to rewrite it into the magic term $magic_B(Z,p(\tilde{x}))$, where $\tilde{x}$ is an eigen-variable standing for the essentially universal x. But we then have to encode that Z is not under the scope of $\tilde{x}$, which would require means out of our setting.

- Some $magic_forall_B(\lambda x{\bullet}Z, \lambda x{\bullet}B(x))$ does catch the scope constraint, but cannot be later unified with $magic_B(a,Y)$ in the clause $B(a,Y) \leftarrow magic_B(a,Y)$ derived from the axiom.

In short, since no scope constraints can be expressed over variables in the head of our l_λ clauses (in particular, heads of clauses may only contain free essentially existential variables), the Magic Set method cannot be directly extended to l_λ.

5.3 Some kind of Magic for l_λ

The obvious solution to the problem of having essentially universal variables in heads of the clauses in the rewritten program is to rewrite without essentially universal variables. Transforming terms containing essentially universal variables into terms containing only essentially existential variables, in a way we will define below, implies a loss of information. Therefore the whole procedure leads to the simulation of a resolution mixing two steps: an approximative SLD resolution which achieves a prediction, and a Bottom-Up evaluation from the restricted set of axioms delimited by the prediction. This procedure was inspired by François Barthélemy's works on mixed resolution strategies [3]. The proof of this result will be given at the end of the section.

Example 7 *Let's consider the following program:*

$p(X,Y) \leftarrow \forall u\ q(Z,Y,u),\ r(X,Z)$

with the axioms

$q(e_1, fX, Y).$

$q(e_1, fX, e_2).$

$r(a, e_1).$

and $r(b,e_1),\ r(b,e_2),\ \dots, r(b,e_n).$

and the query $Query \leftarrow \forall x\ p(a, Hx).$

Then a Bottom-Up evaluation proving from right to left would compute all the facts derived from the $r(b,e_i)$*, which may be numerous and are of no use to prove* $\forall x\ p(a,Hx)$.

To solve this problem, we propose to evaluate the following derived program:

0)	$magic_p(a, H^*)$	*(seed)*
1)	$Success(\forall x\ p(a,Hx)) \leftarrow \forall x\ p(a,Hx)$	*(added clause)*
2)	$p(X,Y) \leftarrow magic_p(X,Y),\ \forall u\ q(Z,Y,u),\ r(X,Z)$	
3)	$magic_q(Z,Y,U^*) \leftarrow magic_p(X,Y),\ r(X,Z)$	*(derived from the first clause)*
4)	$magic_r(X,Z) \leftarrow magic_p(X,Y)$	
5)	$q(e_1,fX,Y) \leftarrow magic_q(e_1,fX,Y)$	
6)	$q(e_1,fX,e_2) \leftarrow magic_q(e_1,fX,e_2)$	
7)	$r(a,e_1) \leftarrow magic_r(a,e_1)$	
8)	$r(b,e_1) \leftarrow magic_r(b,e_1)$	*(derived from the axioms)*
9)	$r(b,e_2) \leftarrow magic_r(b,e_2)$	
	...	
...	$r(b,e_n) \leftarrow magic_r(b,e_n)$	

where Success is the predicate giving the final result.

and H^* *and* U^* *are "predictive" essentially existential variables introduced to stand respectively for* Hx *and* u *which contain essentially universal variables.*

The computed facts are:

i)	$magic_p(a, H^*)$	*(the seed 0)*
ii)	$magic_r(a, Z)$	*(from i and 4)*
iii)	$r(a, e_1)$	*(from ii and 7)*
iv)	$magic_q(e_1, H^*, U^*)$	*(from i, iii and 3)*
v)	$q(e_1, fX, U^*)$	*(from i, iv and 5)*
vi)	$q(e_1, fX, e_2)$	*(from i, iv and 6)*
vii)	$p(a, fX)$	*(from 1 and v)*
viii)	$Success(\forall x\ p(a, fx))$	*(from vii and 0)*

Notice that:

- *the proofs starting from the $r(b, e_i)$ and which are irrelevant for this goal have been ignored in the Bottom-Up evaluation of this Magic Set transformed program.*
- *however some unnecessary facts, like $q(e_1, fX, e_2)$ whose third argument cannot be later unified with an essentially universal x, may be computed due to the inaccuracy introduced by the predictive essentially existential variable U^*.*
- *the computation in its whole is sound and complete.*

We now formalize the method used to transform our program: l_λ clauses are assumed to be written under their expanded form (cf section 3) $D \leftarrow G_1 \ldots G_n$, where the G_i's are universally quantified atomic formulas.

We introduce a mapping μ on a term with a quantifier prefix, which we will sometimes also consider as a substitution on a prefixed term, the following way:

μ is defined by $\mu(M) = \nu^{\{\}}(M)$, where $\nu^{\{\vec{x}\}}$ is defined as follows:

$$\begin{cases} \nu^{\{\vec{x}\}}(\lambda y {\bullet} M) = \nu^{\{\vec{x}y\}}(M) \\ \nu^{\{\vec{x}\}}(Ht_1 \ldots t_p) = \begin{cases} H \text{ if } p = 0 \\ H\nu^{\{\vec{x}\}}(t1) \ldots \nu^{\{\vec{x}\}}(tp) \text{ if } H \text{ is a constant or } H \in \vec{x} \\ H^*\vec{x} \text{ if not, } H^* \text{ being a new essentially existential variable} \end{cases} \end{cases}$$

It is important to notice that μ is not a substitution in the usual sense (respecting a quantifier prefix), since it also transforms essentially universal variables. This mapping μ is likely to perform the "predictive transformation" on terms so that the resulting program may be correctly computed Bottom-Up. Thus no essentially universal variable must remain except those that can be encoded directly in the terms (i.e. the constants and the variables bound by a λ-abstraction). Therefore the resulting terms only contain free essentially existential variables. The basic idea of this mapping is to transform

any essentially universal variable into a new essentially existential one. To respect the condition (#) set on l_λ terms, essentially existential terms will be eliminated and replaced roughly by a new essentially existential variable. In fact, to preserve the correctness of the rewriting procedure, $\{\vec{x}\}$ keeps a trace of all the variables bound by a λ-abstraction up to the current step of decomposition of the term, and these arguments are kept, so that $H^*\vec{x}$ may actually "represent" $Ht_1 \ldots t_p$ (this notion will be formalized later).

We then have the following result:

Theorem 1 (Higher-Order Magic Sets) *Let $\mathcal{P}$ be a l_λ logic program and G be a goal, and let $Magic(\mathcal{P})$ be the program obtained by:*

- $(Success(G) \leftarrow G) \in Magic(\mathcal{P})$
- *if* $(D \leftarrow G_1 \ldots G_n) \in \mathcal{P}$, *then* $(D \leftarrow magic_D,\ G_1 \ldots G_n) \in Magic(\mathcal{P})$
- *if* $(D \leftarrow G_1 \ldots G_n) \in \mathcal{P}$, *then* $(\mu(magic_G_i) \leftarrow magic_D,\ G_1 \ldots G_{i-1}) \in Magic(\mathcal{P})$ *for each* $1 \leq i \leq n$
- $(\mu(magic_G).) \in Magic(\mathcal{P})$

Then a Bottom-Up evaluation of $Magic(\mathcal{P})$ is sound and complete, and mimics a Bottom-Up evaluation from SLD-predicted axioms of $\mathcal{P}$.

Some remarks may be done:

- This theorem is similar in its formulation to the one for first-order Horn Clauses, up to the introduction of the predictive substitution μ. Besides, when applied to Horn clauses terms, μ behaves like the identity substitution, and our theorem restricts to the usual first order Magic Sets method.
- Prediction might also be considered for first-order terms, but is made necessary here because of the impossibility of describing scoping constraints in heads of clause.
- μ is a particular case of predictive substitution. Obviously, replacing each $\mu(magic_G_i)$ by an essentially existential variable also leads to a sound and complete procedure, the difference being that the prediction is even less accurate. Thus a general notion of predictive mapping may be defined, resulting in a more general formulation of the Magic Set method.

To this aim, we introduce here a partial definition of subsumption between prefixed terms:

Definition 1 *$\underline{\exists\vec{y}}\ B$ is said to be subsuming the prefixed term $\underline{Q_A}\ A$ if there exists a substitution σ on the variables $\vec{y}$ of B such that*

$$\sigma(\underline{\exists\vec{y}}\ B) = \underline{Q_A}\ A$$

which stands for the equality under a quantifier prefix:

$$\underline{Q_A\vec{y}'}\ (\sigma(B) = A)$$

where the $\vec{y}'$ are the variables of $\vec{y}$ not instantiated by σ.

Remark: As for first order terms, if $\exists\vec{y}\ B$ subsumes $\underline{Q_A}\ A$, the σ may be obtained by their unification (which writes $Unify(\overline{\underline{Q_A\exists\vec{x}}}\ ,\ B = A))$, when we choose only variables in y to be the ones to be instantiated.

Definition 2 *A mapping of (essentially existential and universal) variables Φ will be called a predictive mapping for a set of prefixed terms if $\Phi(M)$ only contains essentially existential free variables and subsumes M for each term M of the set.*

Of course, mapping on variables canonically extends to mapping on terms, which was implicitly done in the above definition.

The mapping μ defined above is a predictive mapping for the terms of the program P considered. Its corresponding σ may be defined as follows:
for each H^* obtained from a term $Ht_1 \ldots t_p$ by a $\mu^{\{\vec{x}\}}$, $\sigma(H^*) = \lambda\vec{y}\bullet(Ht_1 \ldots t_p)[\vec{x} \leftarrow \vec{y}]$.
This substitution σ actually fits the conditions since it suppresses any essentially universal variable not bound by a λ-abstraction, and respect $\sigma(\mu M) = M$.

Extending the previous theorem to general predictive mapping, we obtain the following result:

Theorem 2 (General Higher-Order Magic Sets) *Let $\mathcal{P}$ be a l_λ logic program, G be a goal, and Φ a predictive mapping for the terms of $\mathcal{P}$,*
and let $Magic(\mathcal{P})$ be the program obtained by:

- $(Success(G) \leftarrow G) \in Magic(\mathcal{P})$
- *if* $(D \leftarrow G_1 \ldots G_n) \in \mathcal{P}$, *then* $(D \leftarrow magic_D,\ G_1 \ldots G_n) \in Magic(\mathcal{P})$
- *if* $(D \leftarrow G_1 \ldots G_n) \in \mathcal{P}$, *then* $(\Phi(magic_G_i) \leftarrow magic_D,\ G_1 \ldots G_{i-1}) \in Magic(\mathcal{P})$ *for each* $1 \leq i \leq n$
- $(\Phi(magic_G).) \in Magic(\mathcal{P})$

Then a Bottom-Up evaluation of $Magic(\mathcal{P})$ is sound and complete, and mimics a Bottom-Up evaluation from SLD-predicted axioms of $\mathcal{P}$.

5.4 Correctness of the general Higher-Order Magic Set method

Magic Sets often look mysterious. A good understanding of this higher-order Magic Set method (as well as first-order Magic Sets) may be obtained using a very general formalism based on *Dynamic Programming* evaluation of Logical Push-Down Automata developed by Bernard Lang for Horn Clauses[8]. In this setting, it appears clearly that the Magic program is nothing but the encoding of the evaluation of the initial program using a specific sound and complete strategy (namely, SLD-evaluating an approximated program and performing exact Bottom-Up resolution on the focused set of axioms).

We sketch here the proof for the theorem concerning the general Higher-Order Magic Set method, derived from this analysis. The result mainly relies on a sound and complete proving

procedure $\mathcal{M}'$, *mixing* Top-Down prediction and Bottom-Up evaluation. As a first intuitive approach, we show the soundness and completeness of a rather similar proving procedure $\mathcal{M}$, where Bottom-Up evaluation *follows* Top-Down prediction. We begin with some simple results:

Lemma 1 *If* $\underline{\exists\vec{y}}\ B$ *subsumes* $\underline{Q_A}\ A$ *then every* Q_A*-closed instance of* A *(i.e. an instantiation of the essentially existential variables in* A *respecting the scoping constraints in* Q_A*) is a* $\underline{\exists\vec{y}}$*-closed instance of* B.

Proof: This lemma is trivially derived from the definition 1 of subsumption.

Lemma 2 *If* $\underline{\exists\vec{y}}\ B$ *subsumes* $\underline{Q_A}\ A$ *then every proof (in the sense of Sequent Calculus) from a* l_λ *logic program* $\mathcal{P}$ *of a* $\underline{Q_A}$*-closed instance of* $\underline{Q_A}\ A$ *is a proof of a closed instance of* $\underline{\exists\vec{y}}\ B$.

Proof: This is straightforward from lemma 1.

This lemma of course applies to a program $\mathcal{P}$ with a predictive mapping Φ. To simplify the writing, we extend canonically such a Φ applying on quantified terms (or atomic formulas) to a mapping over general formulas:

$$\Phi(G_1 \wedge G_2) = (\Phi(G_1)) \wedge (\Phi(G_2))$$

$$\Phi(\forall x\ G) = \forall x\ \Phi(G)$$

The previous lemma then yields the following one:

Lemma 3 *Having a complete proving procedure and a predictive mapping* Φ *for a program* $\mathcal{P}$*, proving* $\Phi(\underline{Q}\ G)$ *from* $\mathcal{P}$ *is complete for proving* $\underline{Q}\ G$ *from* $\mathcal{P}$.

Proof: From lemma 2, we deduce that the set of answers of a program $\mathcal{P}$ for the query $\underline{Q}\ G$ (i.e. of provable $\underline{Q}$-closed instances of G) is contained in the set of answers for the query $\Phi(\underline{Q}\ G)$. Thus answering to $\Phi(\underline{Q}\ G)$ provides us with a complete set of answers for the query $\underline{Q}\ G$.

As a consequence, since SLD-resolution is a sound and complete proving procedure, the SLD-evaluation of $\Phi(\underline{Q}\ G)$ from the program $\mathcal{P}$ is complete for proving $\underline{Q}\ G$. This corresponds to a first step of SLD-prediction $Pred_0$ (i.e. a complete but not necessarily sound resolution) on the goal. This prediction may be extended if we apply this method to each subgoal called by $Pred_0$, which leads to:

Lemma 4 *Considering a program* $\mathcal{P}$*, a goal* G *and a predictive mapping* Φ *for* $\mathcal{P}$*, the method Pred consisting in SLD-proving* $\Phi(G)$ *from the transformed program* $\Phi(\mathcal{P})$*, where all the formulas in the bodies of clause have been transformed by* Φ*, is a complete procedure for proving* G *from the program* $\mathcal{P}$.

This transformed program $\Phi(\mathcal{P})$ implements prediction at each step of an SLD-resolution. The proof may be obtained by induction on the size of the proof tree for an SLD-answer to a given goal. We may then deduce the following result:

Theorem 3 (A sound and complete strategy) *We obtain a sound and complete proving procedure $\mathcal{M}$ for a program $\mathcal{P}$ and a goal G by applying the prediction method Pred followed by a Bottom-Up computation from the axioms involved in the prediction.*

Proof: By lemma 4, the prediction $\mathcal{M}$ is complete for proving G and thus guarantees that there is no Bottom-Up proof for G using an axiom not involved in $\mathcal{M}$. This gives the completeness of the method $\mathcal{M}'$. Soundness is obtained by applying the usual Bottom-Up proving procedure to the program $\mathcal{P}$, starting only from the relevant axioms.

This proving procedure may be refined by mixing the prediction and the Bottom-Up resolution, instead of applying them successively: each time a predictive subgoal $\Phi(G)$ is proved, a Bottom-Up step is computed trying to unify the predicted fact with G. In case of success, the substitution thus obtained is transmitted to the remaining predictive subgoals, thus restricting even more the search space. This strategy may be viewed as an extension of Earley Deduction [5].
The sets of subgoals and facts may be defined by the following mutually recursive formulas, derived from Nilsson's simplified expression of Carnegie Mellish's work [14]:

$$Call = Init \cup \bigcup_{A_0 \leftarrow \Phi(G_1),\ldots,\Phi(G_i),\ldots,\Phi(G_n) \in \Phi(\mathcal{P})} \{\theta\Phi(G_i) \mid B_0 \in Call,\ B_1,\ldots,B_{i-1} \in Succ \text{ and } mgu(A_0 \ldots G_{i-1}, B_0 \ldots B_{i-1}) = \theta \neq \bot\}$$

$$Succ = \bigcup_{A_0 \leftarrow G_1,\ldots,G_n \in \mathcal{P}} \{\theta(A_0) \mid B_0 \in Call,\ B_1,\ldots,B_n \in Succ \text{ and } mgu(A_0 \ldots G_n, B_0 \ldots B_n) = \theta \neq \bot\}$$

Init contains the initial goals, and *Succ* is initialized with the axioms of $\mathcal{P}$.

Theorem 4 (Another sound and complete strategy) *We obtain a sound and complete proving procedure $\mathcal{M}'$ for a program $\mathcal{P}$ and a goal G by applying the prediction method Pred mixed with a Bottom-Up computation as described above.*

Before we attack the proof of this theorem, we need the following lemma:

Lemma 5 *If $\underline{\exists\vec{y}}\ B$ subsumes $\underline{\exists\vec{u}\forall\vec{v}}\ A$, then for each $\underline{\exists\vec{u}\forall\vec{v}}$-substitution τ whose substitution terms does not contain essentially universally variables quantified in a prefix, $\tau(\underline{\exists\vec{y}}\ B)$ subsumes $\tau(\underline{\exists\vec{u}\forall\vec{v}}\ A)$.*

This property concerns interesting particular cases of subsumption, since the quantified terms $\underline{\exists\vec{u}\forall\vec{v}}\ A$ are those representing quantified atomic goals in l_λ. The τ concerned are the restrictions to the variables of A of the unifiers obtained by chaining A with a clause head.
Proof: since the substitution terms only contain essentially existential variables, $\tau(\underline{\exists\vec{y}}\ B)$ appears under the form $\underline{\exists\vec{y}'}\ B'$, so it is correct to consider our (partial) subsumption.
The proof of this lemma relies on an induction on the structure of the term B: we modify σ such that $\sigma(\underline{\exists\vec{y}}\ B) = \underline{\exists\vec{u}\forall\vec{v}}\ A$ to obtain σ' such that $\sigma'\tau(\underline{\exists\vec{y}}\ B) = \tau(\underline{\exists\vec{u}\forall\vec{v}}\ A)$

- if B is a (necessarily essentially existential) variable,
  - either B occurs in A, then necessarily $A = B$, so $\tau(A) = \tau(B)$, and no σ' is needed.
  - or B does not occur in A, then B is not instantiated by τ, and we choose $\sigma'(B) = \tau(M)$ if $\sigma(B) = M$.
- if B is a functional term
  - if B is essentially universal, then its head symbol must be a constant or a variable λ-abstracted before. In both cases, A must also be a functional term with the same head symbol. We may then apply the induction hypothesis to the arguments.
  - if B is essentially existential, then the condition (#) states that the arguments in B are essentially universal variables quantified in the scope of the head of B: $B = uy_1 \dots y_n$. In this case, the y_i are necessarily previously λ-abstracted variables. The discussion is then similar to the case where B is a variable: if u occurs in A then u is not modified by σ and thus we must have $A = B$. If it does not, then we choose $\sigma'(u) = \lambda x_1 \dots x_n \bullet \tau(A)$.
- if B is a λ-abstraction $\lambda x \bullet B'$, then so must be A, say $A = \lambda y \bullet A'$.
  We apply the induction hypothesis to B' and $A'[y \longmapsto x]$, where x is considered as an essentially universal variable quantified in the scopes of the previous variables.

Proof of theorem 4: The soundness of the procedure is obvious since facts are proved by genuine Bottom-Up computations (the method $\mathcal{M}'$ only leads these computations). We prove the completeness of $\mathcal{M}'$ the following way: considering a program $\mathcal{P}$, a predictive mapping Φ, we note $\Phi(\mathcal{P})$ the program obtained by applying Φ to the clause bodies, and we reason by induction on the size S of an SLD-proof tree for an answer ρ to a goal G:

> If $S = 1$, then $\rho(G)$ is an instance of an axiom A of $\mathcal{P}$. This axiom is also in $\Phi(\mathcal{P})$, since only bodies of clause are modified. So, by lemma 3, we deduce $\rho(G)$ as an answer for the predictive goal $\Phi(\underline{Q}\ D)$. As $\rho(G)$ also unifies with $\underline{Q}\ G$ in the Bottom-Up step, it appears as a $\mathcal{M}'$-answer for $\underline{Q}\ G$.
>
> Induction hypothesis: we assume the result to be true up to S: for each goal with an answer whose SLD-proof tree has a size inferior or equal to S, applying the method $\mathcal{M}'$ with a predictive mapping Φ is complete.
> We consider now a goal G for which an answer ρ exists, whose SLD-proof tree has the size $S + 1$. We separate the cases according to the last rule used in the proofs. Since l_λ (as a restriction of L_λ) supports uniform proofs [12], this last rule is a right-introduction rule until we deal with an atomic goal. For l_λ, this restricts to the following cases:
>
> - if we use a right-$\wedge$-Introduction (corresponding to the AND case of the SLD interpreter), then $\underline{Q}\ G$ is of the form $(\underline{Q_1}\ G_1) \wedge (\underline{Q_2}\ G_2)$, and $\mathcal{P}$ provides us with proofs for $\rho(G_1)$ and $\rho(G_2)$.
>   Using $\mathcal{M}'$, we first try to prove $\underline{Q_1}\ G_1$, and we consider the predictive goal $\Phi(\underline{Q_1}\ G_1)$. Applying the induction hypothesis, we deduce the completeness of

the procedure for this goal, and thus that some answer τ is computed by $\mathcal{M}'$, such that $\rho(G_1)$ is a $\underline{Q_1}$-instance of $\tau(G_1)$. $\rho(G_1)$ is thus a $\mathcal{M}'$-answer for $\underline{Q_1}\ G_1$.
The procedure $\mathcal{M}'$ then considers the predictive goal $\rho\Phi(\underline{Q_2}\ G_2)$. By lemma 5, $\rho\Phi(\underline{Q_2}\ G_2)$ subsumes $\rho(\underline{Q_2}\ G_2)$. We may then apply the induction hypothesis to the goal $\underline{Q_2}\ \rho(G_2)$, with the predictive mapping $\Phi :: [\rho(G_2) \mapsto \rho\Phi(G_2)]$. The completeness obtained guarantees that $\rho(G_2)$ also constitutes a $\mathcal{M}'$-answer for $\underline{Q_2}\ G_2$. Thus $\tau(G)$ is also a $\mathcal{M}'$ answer to G.

- since we assume that the quantifiers have been distributed to atomic goals, as was suggested in 3.2, right-$\forall$-introductions (corresponding to the AUGMENT rule of the SLD-interpreter) are always followed by left-$\Rightarrow$-introduction (corresponding to the BACKCHAIN rule of the SLD-interpreter). One unique proof may encompass this case and that of only using the BACKCHAIN rule, because it may be considered as a BACKCHAINing over quantified formulae (using the unification over quantified terms presented in 2.3). This proof stands as follows:
  In the SLD-proof, $\underline{Q}\ G$ is first unified with some head A of a clause $A \leftarrow G'$. Using $\mathcal{M}'$, we consider the predictive goal $\Phi(\underline{Q}\ G)$. By lemma 3, considering $\Phi(\underline{Q}\ G)$ is complete for proving $\underline{Q}\ G$. So $\underline{Q}\ G$ also unifies with A via σ, and $\rho(G')$ is an SLD-answer to $\sigma(G')$.
  Besides, by lemma 5, $\sigma\Phi(G')$ subsumes $\sigma(G')$. Therefore, we may apply the induction hypothesis to $\sigma(G')$ with the predictive mapping $\Phi :: [\sigma(G') \mapsto \sigma\Phi(G')]$, and deduce the completeness of proving $\sigma(G')$. As a consequence, $\rho(G')$ is also a $\mathcal{M}'$-answer to $\sigma(G')$.
  So $\rho(G)$ appears as an answer for the predictive goal $\Phi(\underline{Q}\ G)$. A last Bottom-Up step is then computed by $\mathcal{M}'$, unifying the facts obtained with $\underline{Q}\ G$. This step of course preserves the completeness of proving $\underline{Q}\ G$, and $\rho(G')$ may be chained back to $\underline{Q}\ G$, giving $\rho(G)$ as a $\mathcal{M}'$-answer to $\underline{Q}\ G$.

The General Higher-Order Magic Sets theorem may be deduced by showing that the Magic Set method is actually a Bottom-Up implementation of this resolution procedure $\mathcal{M}'$: the call of subgoals in the Top-Down predictive phase is encoded into the predicates *magic_X*. The proof is straightforward.

A better and more general proof may be obtained, as we said, using the formalism of LPDA: compiling $\mathcal{P}$ with the logic $\mathcal{M}'$ and a Bottom-Up control yields the general higher-order Magic Set method. This view is unfortunately too long to present here.

6 Conclusions

In the purpose of extending the possible use of L_λ to some application fields, we have studied a restriction l_λ which allows function variables and universal quantification. We showed that a Bottom-Up strategy is available for this language. This strategy is sound and complete, and very simple, since the unification takes care of all the higher-order features.
Bottom-Up approaches present the advantages of being complete, and suitable for parallel execu-

tion. They are also sometimes more adapted to the kind of reasoning used in the concerned domain. For instance, this restriction l_λ suffices to implement semantic interpretation [16], the treatment of (linear) implication being explicitly described in the program. This example is typically a case of a program containing a left-recursive clause, and for which Bottom-Up is more efficient: an SLD-resolution will arbitrarily often apply the rule for discharging assumptions, even though there may be not enough assumption introductions in the tree below. Conversely, in a Bottom-Up resolution, assumptions will be introduced before they are discharged.

As an application, we have studied the possibility of extending the Magic Set method developed for first-order Horn Clauses [17] to the higher-order language l_λ. This method solves the problem of Bottom-Up approaches of computing many unnecessary facts by simulating an SLD-resolution by a Bottom-Up evaluation of a transformed program. We showed that direct extension was not possible. But, provided some loss of information in an SLD-prediction, we may propose an adaptation of the method to l_λ, which restricts the space of search all the more significantly as the prediction may be accurate. This higher-order Magic Set method also relies on the Bottom-Up computation of a program which may be deduced from the original one by a simple transformation. It may be applied with different predictive mappings. Yet, there is no way to obtain an exact prediction since scoping constraints cannot be expressed in heads of clause. This therefore represents an obligatory inaccuracy which notably occurs in the following cases:

- an essentially universal variable is replaced by an essentially existential one, which may then be instantiated during the prediction.
- an essentially existential variable appearing both in the head and the body of a clause may be replaced by another one in the body (for instance if we had $(X\,y)$, with y universally quantified in the body). The direct correlation between the two variables is lost in the prediction.

The problem of optimizing this prediction remains open to discussions and refinements, and we may ask just how much this Higher-Order Magic Set method may be really interesting. A first theoretical answer is that, although the prediction phase cannot be made exact, this method actually leads to a reduction of the facts computed. Implementation and tests are needed for a complete answer.
Anyway we believe that at least this theoretical setting may represent a first step towards developing L_λ to extend first-order Horn Clauses applications.

7 Extensions

The presentation we made here both of Bottom-Up evaluation and of some Magic Set transformation concerns the language l_λ. We consider now the possibility of extending these results to larger sets of terms and formulas:

- Extending the terms: the unification involved in l_λ makes use of a restricted form of β-conversion, namely β_0-conversion [10]. This is the reason for its good properties. Now full

$\beta\eta$-unification may in fact be considered since the Bottom-Up procedure only relies on the property of essentially existential variables in a clause, whether in the head or in the body, to be out of the scopes of the essentially universal ones. Bottom-Up resolution could then also apply to the same set of formulas, where the condition (#) on the essentially existential terms is removed. It even makes the predictive mapping in the Magic Set method easier to be found, since essentially universal variables may be directly transformed into essentially existential ones (lemma 5 would yet have to be proved again with this extension, if we want to guarantee the completeness of the Higher-Order Magic Set method).
On the other hand, full $\beta\eta$-unification is only semi-decidable, and the completeness gained with a Bottom-Up resolution may be lost in such a unification.

- Extending the logic: to reach the whole logic in HOHH, we first need to allow embedded mixed quantifications in the clause bodies. The existential quantification (*pi*) is not authorized in L_λ since it may introduce terms offending the condition (#). If we extend our terms as described above, we may add this predicate *pi* provided we (statically) skolemize all the clause bodies. If not, we would have essentially existential variables under the scope of an essentially universal one, and for example, it would be harder to treat a goal such as $\forall x \exists y\ (P(x,y) \wedge Q(x,y))$.

  The main difficulty is then to include embedded implications. Such a feature is problematic since it requires introducing and discharging assumptions, and thus retaining some "history" for each fact computed. Some solution may perhaps be obtained by extending the Magic Sets prediction to encompass implication and guide the Bottom-Up search. The transformation would then need to be applied dynamically. We must admit we have not explored this perspective yet.

8 Acknowledgements

Thanks to François Barthélemy for his helpful reading and comments, to Eric Villemonte de la Clergerie for enlightening discussions. I am also grateful to the workshop reviewers who provided me with very enriching comments.

References

[1] F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *SIGMOD, invited paper*, 1986.

[2] F. Bancilhon and R. Ramakrishnan. *Performance Evaluation of Data Intensive Logic Programs.* Morgan Kaufman, 1988.

[3] François Barthélemy. Prédire à bon escient. In *Actes JTASPEFL'91*, Bordeaux (FRANCE), Octobre 1991.

[4] C. Beeri and R. Ramakrishnan. On the power of magic. In *Proceedings of the 6th Symposium on Principles of Database Systems*, pages 269–283, 1987.

[5] Jay Earley. An efficient context-free parsing algorithm. *Communications A.C.M.*, 13(2):92–102, February 1970.

[6] Gérard Huet. A unification algorithm for typed λ–calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[7] Alain Hui Bon Hoa. A simple abstract interpreter for a higher-order logic programming language. In *Proceedings of the* 4th *International Symposium, PLILP 92*, August 92.

[8] Bernard Lang. Complete evaluation of Horn Clauses: an automata theoretic approach. Research Report 813, INRIA, November 1988.

[9] Dale Miller. Unification under a mixed prefix. to appear in the Journal of Symbolic Computation.

[10] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1:497–536, 1991.

[11] Dale Miller and Gopalan Nadathur. Some uses of higher-order logic in computational linguistics. In *Proceedings of the* 24th *Annual Meeting of the Association for Computational Linguistics*, 1986.

[12] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[13] Gopalan Nadathur. *A Higher-Order Logic as the basis for Logic Programming.* PhD thesis, University of Pennsylvania, 1987.

[14] Ulf Nilsson. Abstact interpretation: A kind of magic. In *Proceedings of PLILP91*, 1991.

[15] F.C.N. Pereira and D.H.D. Warren. Parsing as deduction. In *Proceedings of the 21st Annual Meeting of the Association for Computationnal Linguistic*, pages 137–144, Cambridge (Massachussetts), 1983.

[16] Fernando C. N. Pereira. Semantic interpretation as higher-order deduction. In Springer-Verlag, editor, *Lecture Notes in Artificial Intelligence*, pages 78–96, 1990.

[17] Raghu Ramakrishnan. Magic templates: A spellbounding approach to manipulating formulas and programs. In *Proceedings of the* 5th *International Conference/Symposium on Logic Programming*, pages 140–159, 1988.

[18] Anthony Rich and Marvin Solomon. A logic–based approach to system modelling. Extended Abstract, 1990.

[19] François Rouaix. private communication.

[20] Eugene J. Rollins and Jeannette M. Wing. Specifications as search keys for software libraries. In Koichi Furukawa, editor, *Proceedings of the* 8th *International Conference on Logic Programming*, pages 173–187, 1991.

[21] K. Uehara, R. Ochitani, O. Kakusho, and J. Toyoda. A bottom-up parser based on predicate logic: A survey of the formalism and its implementation technique. In *Proc. of the International Symposium on Logic Programming*, pages 220–227, Atlantic City, 1984.

An Instruction Set for Higher-Order Hereditary Harrop Formulas (Extended Abstract)

Keehang Kwon and Gopalan Nadathur [1]
Department of Computer Science
Duke University Durham, NC 27706, USA
kwon@cs.duke.edu, gopalan@cs.duke.edu

We have been investigating methods for efficient implementation of the logic of hereditary Harrop formulas. There are several similarities in the structure of this logic and the logic of Horn clauses that have convinced us of the wisdom of using a WAM-like model as the basis for our work. However, the logic of interest extends Horn clause logic in several significant respects and methods for dealing with these have to be developed. In particular, four facets of the logic can be identified for which new implementation techniques have to be devised:

(1) the presence of the two new primitives. GENERIC and AUGMENT, for controlling the pattern of search,

(2) the presence of lambda terms and the need to perform lambda conversion on these terms,

(3) the embedding of higher-order unification with its branching characteristic within the normal Prolog computation regime, and

(4) the use of polymorphic typing that. within logic programming, lead to a need for processing types at run-time.

We have developed a sequence of schemes for dealing with these new features that, in our opinion, fit gracefully into the general structure of the WAM [4, 9, 10, 12]. In each of these efforts, we have focused on one specific aspect and described the mechanisms, usually in addition to those already present in the WAM, for implementing that aspect. The purpose of the paper being described is to consolidate these various discussions into one abstract machine that implements the entire logic of hereditary Harrop formulas; as such, it serves as a blueprint for an implementation that is currently being carried out. In this extended abstract we outline only the problems and the broad method of treatment. The full paper will contain a detailed description of the components of this machine and its complete instruction set.

The search primitive GENERIC arises from the inclusion of universal quantifiers in goals. The operational semantics of this logical symbol involves introducing a new constant and then solving the goal resulting from instantiating the quantifier with this constant. This interpretation cannot be implemented exactly as described because of the presence of existential quantifiers. The latter involves guessing an appropriate instance. and the only reasonable implementation is to postpone

[1] Work on this paper has been supported by the NSF grant CCR-89-05825.

the guessing till it can be determined through unification. The problem then is that when a guess is made, it might violate the newness constraint on the constant used for universal quantifiers. As a concrete example, it should not be possible to solve the goal $\exists x \forall y p(x,y)$ from the program $\{\forall x p(x,x)\}$. The technique generally used to deal with this problem is to skolemize the universal quantifiers before attempting to solve the goal. However, a static skolemization will not work in the context of hereditary Harrop formulas. As an example, the goal $((\forall x p(x) \supset q) \supset \exists y(p(y) \supset q))$ must not succeed, but would succeed under the usual understanding of the static skolemization process. A dynamic form of skolemization can be used and several related methods for solving this problem have been outlined in [7]. However, these methods do not blend easily into the design of an abstract machine and a compilation scheme. Fortunately, there is a method that is readily implementable. This method (discussed in [1] and [3] and proved correct in [8]) involves thinking of a hierarchy of "Herbrand universes" and tagging variables and constants based on the universe they belong to. The tag on a variable indicates that it can be instantiated only by a term belonging to the universe at that level. The tags thus constrain unification and conspire to ensure the correctness of bindings. From the perspective of our machine, tags are easily representable as an extra field with variables and constants. Universal and existential quantifiers compile into simple instructions that set tags for variables and possibly increment a universal tag index. The checking of tags blends readily into the compiled code generated for unification — the instructions (for the first-order case) remain the same but possibly involve a simple additional operation. The interpretive phase of unification (embodied in the `unify_value` instruction in the WAM) involves a check for tag compatibility when a variable is ultimately bound. However this can be incorporated into the "occurs-check" that the WAM must do to ensure correctness. (Just as in the WAM, situations can be described where this check may be elided).

The AUGMENT primitive arises from permitting implications in goals. The operational semantics of this symbol is as follows: to solve the goal $D \supset G$, we add D to the program (the syntax of D is restricted for this to be possible) and then attempt to solve G. From the perspective of providing a reasonable implementation of this operation, there are three issues to be dealt with. First, we have to deal with changing sets of program clauses. For example, solving $(D_1 \supset G_1) \wedge (D_2 \supset G_2)$ from a program $\mathcal{P}$ involves using programs $\mathcal{P}$, $\mathcal{P} \cup \{D_1\}$ and $\mathcal{P} \cup \{D_2\}$. A reasonable means for managing these different program contexts — such as creating each one by adding and removing parts of code — is necessary. Second, we would like to compile (and share compiled code for) program clauses that appear on the left of implications. This requirement is complicated by the fact that slightly different versions of a program clause may be needed at different points. For example, consider using the program clause $\forall x(((D(x) \supset G) \wedge p(x)) \supset p(f(x)))$ for solving $\exists y p(f(f(y)))$, assuming p is a predicate name, f is a function symbol, D is some program clause and G is a goal. (We assume that a program clause is provided for p for the base case of the recursion). Now two clauses will need to be added to the program: $D(f(y))$ and $D(y)$ in the course of solving the query. There is nevertheless a considerable amount of structure that is common between these two clauses and we would like our implementation to permit this to be shared: this is essential if we are to compile the code for D in any sense. The final problem deals with backtracking. Consider solving a goal such as $\exists x((D_1 \supset G_1(x)) \wedge G_2(x))$. Assume that we have succeeded in solving the goal $(D_1 \supset G_1(x))$. However, the instantiation determined for x is such that the attempt to solve

$G_2(x)$ fails. We then have to backtrack to trying to find another solution to $D_1 \supset G_1(x)$. Within the WAM framework, this involves returning to some subgoal of $G_1(x)$. Notice, however, that the program in existence at that point has to be reconstructed. Some simple and efficient means for doing this is needed.

Our machine embodies a solution to all these problems posed by AUGMENT. The problem with slightly different versions of program clauses is solved by using the idea of a closure: a program clause is represented by code and bindings for variables. The bindings are determined by some specified environment record in the sense of the WAM. The compiled code for the clause contains initialization instructions that work relative to this environment record. Mechanisms are included for making the appropriate environment record available when the code is to be executed. The changing program contexts are realized by using a stack based representation of available program clauses. The compiled code for an implication gives rise to an *implication point record* on the local stack. The implication point record adds clauses essentially by defining a new access function to clauses available at the point of its creation. Some work has to be done in order to set up this record at run-time, but a considerable amount of the task can be compiled. The action with regard to backtracking is simply to resurrect an earlier access function. The usual WAM devices serve to determine whether or not an access function will be required subsequent to a successful computation, preserving the scheme for reclaiming parts of the local stack. (The overall scheme combines ideas in [3] and [5] and is described completely in [9]).

Given that lambda terms are a central part of the logic of higher-order hereditary Harrop formulas, an efficient implementation requires a good representation to be devised for these terms. In determining what is a good representation, a distinction must be made between a situation where these terms are used as a means for computing as in functional programming languages and where they are used as data structures. In the latter case the representation must make the structures of terms readily apparent. Further, the ability to determine equality or unifiability modulo lambda conversion should be supported. In particular, it should be easy to ascertain whether two terms are identical except for a difference in bound variable names and the operation of β-reduction on terms should also receive an efficient implementation. In our context, the latter aspect dictates a representation that allows substitutions to be performed lazily. Thus, consider the task of determining whether the terms $(\lambda x \lambda y \lambda z((x\ y)\ s))\ (\lambda w w)$ and $(\lambda x \lambda y \lambda z((x\ z)\ t))\ (\lambda w w)$ are equal, assuming that s and t are complex terms. It may be concluded that they are not, by observing that these terms reduce to $(\lambda y \lambda z(y\ s'))$ and $(\lambda y \lambda z(z\ t'))$, where s' and t' result from s and t by appropriate substitutions. Notice that it is not really necessary to determine the exact form of s' and t' before reaching this conclusion, and a means for performing substitutions lazily can save a potentially costly operation. In implementing this idea, the notion of environments from functional programming can be used. However, the details of such a scheme are considerably more intricate here because, as is clear from the example considered, reductions may have to be done embedded within abstractions and substitutions must also be percolated into such contexts. A scheme has been worked out that takes these factors into account and also makes the checking of α-convertibility easy by being based on de Bruijn's nameless representation for lambda terms [12]. Our machine embodies a version of this representation.

The notion of unification that is pertinent to higher-order hereditary Harrop formulas is based

on equality modulo λ-conversion. The resulting computation is quite different from that in Prolog, particularly in that most general unifiers do not exist anymore. A procedure for finding unifiers has been described by Huet [2]. This procedure has two phases that are applied repeatedly. One of these phases simplifies the structure of the terms to be unified, eventually either determining that no unifiers can exist or producing a set of pairs of terms whose unifiers are identical to the unifiers of the initial pair. In the latter case, the set produced is one for which a unifier can be readily provided, *i.e.* it is a *solved* set, or one of a finite number of possibilities may be tried to progress the search towards finding a unifier. From an implementation prespective, the structure of this procedure dictates that sets of pairs of terms that have to be unified, the so-called *disagreement* sets, have to be represented explicitly. The representation must satisfy certain characteristics to yield an efficient implementation. One requirement arises from the fact that disagreement sets change incrementally as unification proceeds, with large parts being preserved between sets. Thus a representation that exhibits a large amount of sharing between sets is desirable. Another requirement is that, in light of backtracking, it should be possible to reinstate previous sets rapidly. Our machine embodies a scheme for maintaining disagreement sets that appears to meet these criteria. In essence the scheme maintains a stack of disagreement pairs and a linked list through the stack indicates the "current" disagreement set. Reinstatement of a previous set upon backtracking is facilitated by making the list doubly linked and using a trailing mechanism that is in several respects similar to that used in Prolog implementations for resetting state. Another requirement that Huet's procedure imposes is the ability to handle branching within unification. This is catered to within our machine by conducting a depth-first search, using a *branch point* record to encode the alternatives that are as yet unexplored in its state. These new records are akin to the choice point record of the WAM and similarly enable a rapid return to an earlier state followed by the choice of an alternative search path. Finally, although branching in unification may eventually be necessary, experimental evidence suggests that it might often be avoided by some simple processing steps [6]. Our implementation is sensitive to this fact at several levels. First, the processing structure permits the easy application of such steps. Second, the creation of branch point records and the explicit encoding of disagreement sets is delayed until *after* these steps have been applied. Third, specific operations are considered towards eliminating branching. With regard to the last aspect, our implementation permits a treatment of first-order like unification problems through the usual mechanisms of the WAM and can deal with these problems almost entirely through compiled code.

The last issue pertains to typing. It may at first seem somewhat intriguing that types should play a role in determining the run-time support of a language. The reason for this, as discussed in [11], is twofold: the behavior and outcome of the unification process is influenced by the types of various expressions and, because of a polymorphism that is permitted in the language, the actual types involved are only known in the course of execution. Now, it is desirable to reduce the runtime processing of types to the greatest possible extent in a good implementation. A look at the typing regimen used in conjunction with hereditary Harrop formulas shows that a clever representation of types and a careful use of information present during compilation can considerably reduce the time and space required for type analysis. The essential idea is that by virtue of type declarations a "skeleton" is known for the type of every primitive symbol at runtime and this skeleton can be shared across several incarnations of the symbol. Further, it is actually possible to compile the type

analysis that is required due to the refinement to "leaves" in this skeleton in parts of the program. This type analysis is in fact a form of first order unification that the WAM machinery is adept at carrying out. A proper meshing of the unification instructions for types with that for terms is required (involving answering questions such as when type comparison must be initiated and when types have to be written as opposed to checked for compatibility). These details have been worked out and are embodied in our machine. At a level of detail, this requires the addition of a heap, called a type heap, for the processing of types in our machine. These can be merged into the usual heap. However their separation adds a desirable flexibility to the processing scheme.

References

[1] Conal Elliott and Frank Pfenning. A semi-functional implementation of a higher-order logic programming language. In Peter Lee, editor, *Topics in Advanced Language Implementation*, pages 289–325. MIT Press, 1991.

[2] Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[3] Bharat Jayaraman and Gopalan Nadathur. Implementation techniques for scoping constructs in logic programming. In Koichi Furukawa, editor, *Eighth International Logic Programming Conference*, pages 871–886, Paris, France, June 1991. MIT Press.

[4] Keehang Kwon, Gopalan Nadathur, and Debra Sue Wilson. Implementing polymorphic typing in a logic programming language. Submitted, August 1992.

[5] Evelina Lamma, Paola Mello, and Antonio Natali. The design of an abstract machine for efficient implementation of contexts in logic programming. In G. Levi and M. Martelli, editors, *Sixth International Logic Programming Conference*, pages 303–317, Lisbon, Portugal, June 1989. MIT Press.

[6] Spiro Michaylov and Frank Pfenning. An empirical study of the runtime behavior of higher-order logic programs. In *Conference Record of the Workshop on the λProlog Programming Language*, Philadelphia, July-August 1992.

[7] Dale Miller. Unification under a mixed prefix. Technical Report MS-CIS-91-81, Computer Science Department, University of Pennsylvania, October 1991. To appear in the *Journal of Symbolic Computation*.

[8] Gopalan Nadathur. A proof procedure for the logic of hereditary Harrop formulas. Technical Report CS-1992-17, Department of Computer Science, Duke University, November 1992. To appear in the *Journal of Automated Reasoning*.

[9] Gopalan Nadathur, Bharat Jayaraman, and Keehang Kwon. Scoping constructs in logic programming: Implementation problems and their solution. Submitted, May 1992.

[10] Gopalan Nadathur, Bharat Jayaraman, and Debra Sue Wilson. Implementation considerations for higher-order features in logic programming. Submitted, November 1992.

[11] Gopalan Nadathur and Frank Pfenning. The type system of a higher-order logic programming language. In Frank Pfenning, editor, *Types in Logic Programming*, pages 245–283. MIT Press, 1992.

[12] Gopalan Nadathur and Debra Sue Wilson. A representation of lambda terms suitable for operations on their intensions. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 341–348. ACM Press, 1990.

Implementing a Notion of Modules in the Logic Programming Language λProlog

Keehang Kwon, Gopalan Nadathur and Debra Sue Wilson [1]
Department of Computer Science
Duke University
Durham, NC 27706, USA
kwon@cs.duke.edu, gopalan@cs.duke.edu, dsw@cs.duke.edu

1 Abstract

Issues concerning the implementation of a notion of modules in the higher-order logic programming language λProlog are examined. A program in this language is a composite of type declarations and procedure definitions. The module construct that is considered permits large collections of such declarations and definitions to be decomposed into smaller units. Mechanisms are provided for controlling the interaction of these units and for restricting the visibility of names used within any unit. The typical interaction between modules has both a static and a dynamic nature. The parsing of expressions in a module might require declarations in a module that it interacts with, and this information must be available during compilation. Procedure definitions within a module might utilize procedures presented in other modules and support must be provided for making the appropriate invocation during execution. Our concern here is largely with the dynamic aspects of module interaction. We describe a method for compiling each module into an independent fragment of code. Static interactions prevent the compilation of interacting modules from being completely decoupled. However, using the idea of an interface definition presented here, a fair degree of independence can be achieved even at this level. The dynamic semantics of the module construct involve enhancing existing program contexts with the procedures defined in particular modules. A method is presented for achieving this effect through a linking process applied to the compiled code generated for each module. A direct implementation of the dynamic semantics leads to considerable redundancy in search. We present a way in which this redundancy can be controlled, prove the correctness of our approach and describe run-time structures for incorporating this idea into the overall implementation.

2 Introduction

This paper concerns the implementation of a notion of modules in the logic programming language λProlog. Logic programming has traditionally lacked devices for structuring the space of names and procedure definitions: within this paradigm, programs are generally viewed as monolithic collections of procedure definitions, with the names of constants and data constructors being implicitly defined and visible everywhere in the program. Although the absence of such facilities is not seriously felt

[1] Work on this paper has been supported by the NSF grant CCR-89-05825.

in the development of small programs, structuring mechanisms become essential for programming-in-the-large. This fact has spurred investigations into mechanisms for constructing programs in a modular fashion (*e.g.*, see [11, 14, 19, 20]) and has also resulted in structuring devices being included in some implementations of a Prolog-like language on an *ad hoc* basis. Most proposals put forth have, at the lowest level, been based on the use of the logic of Horn clauses. This logic does not directly support the realization of structuring devices, and consequently these have had to be built in at an extra-logical level. The logic of hereditary Harrop formulas, a recently discovered extension to Horn clause logic [13], is interesting in this respect because it contains logical primitives for controlling the visibility of names and the availability of predicate definitions. The language λProlog is based on this extended logic and thus provides logical support for several interesting scoping constructs [10, 11]. The notion of modules whose implementation we describe in this paper is in fact based on these new mechanisms.

The language λProlog is in reality a typed language. One manifestation of this fact is that programs in this language consist of two components: a set of type declarations and a set of procedure definitions. The module concept that we consider is relevant to a structuring of programs with respect to both components. In a simplistic sense, a module corresponds to a named collection of type declarations and procedure definitions. This view of modules reveals that the use of this structuring notion has both static and dynamic effects. The typical use that might be expected of any module is that of making it contents available in some fashion within a program context such as another module. The main impact of making the declarations in a module visible must clearly be a static one: to take one example, the type associated with some constant by the module in question may be needed for parsing expressions in the new context. The effect with regard to predicate definitions is, on the other hand, largely dynamic. Thus, procedure definitions in the new context might contain invocations to procedures defined in the "imported" module. The important question to be resolved, then, is that of how a reference to code is to be resolved in a situation where the available code is changing dynamically.

From the perspective of implementing the module notion, the main concern is really with the dynamic aspects. In particular, our interest is largely in a method for compiling the definitions appearing in modules and in the run-time structures needed for implementing the prescribed semantics for this construct. We examine these questions in detail in this paper and suggest solutions to them. Now, λProlog has several new features in comparison with a language such as Prolog and a complete treatment of compilation requires methods to be presented for handling these features as well. We have studied the implementation issues arising out the other extensions in recent work and have detailed solutions to them [7, 16, 17]. We outline the nature of these solutions here but do not present them in detail. In a broad sense, our solutions to the other problems can be embedded in a machine like the Warren Abstract Machine (WAM) [21]. We start with this machine and describe further enhancements to it that serve to implement the dynamic aspects of the module notion. There are several interesting characteristics to the scheme we ultimately suggest for this purpose, and these include the following:

(i) A notion of separate compilation for modules is supported. As we explained above, there is a potential for static interaction between modules that makes completely independent compilation impossible. However, this situation is no different from that in any other programming

language. We propose the idea of an interface definition to overcome this problem. Relative to such definitions, we show that the separate compilation goal can actually be achieved.

(ii) A notion of linking is described and implemented. The dynamic use of modules effectively reduces to solving goals of the form M ==> G where M is a module name. The expected action is to enhance an existing program context with the definitions in M before solving G. The symbol ==> can, in a certain sense be viewed as a primitive for linking the compiled code generated for a module into a program context. Using ideas from [8] and [16] we show how this primitive can be implemented.

(iii) A method for controlling redundancy in search is described. The dynamic semantics presented for modules in [11] can lead to the definitions in a module being added several times to a program context, leading to considerable redundancy in solving goals. We present a sense in which this redundancy can be eliminated, prove the correctness of our approach and show how this idea can be incorporated into the overall implementation. The general idea in avoiding redundancy has been used in earlier implementations of λProlog [2, 9]. However, ours is, to our knowledge, the first proof of its correctness and the embedding of the idea within our compilation model is interesting in its own right.

The remainder of this paper is structured as follows. We describe the language of λProlog without the module feature in the next section, focussing eventually on the general structure of an implementation for this "core". In Section 4, we present the module notion that is the subject of this paper and outline the main issues in its implementation. In Section 5, we present our first implementation scheme. This scheme permits separate compilation and contains the run-time devices needed for linking. However, it has the drawback that it is may add several copies of a module to a program context leading to the mentioned redundancy in search. We discuss this issue in detail in Section 6 and show a way in which redundancy can be controlled. In Section 7 we use this idea in describing mechanisms that can be incorporated into the basic scheme of Section 5 to ensure that only one copy of a module is available in a program context at any time. Section 8 concludes the paper.

3 The Core Language

We describe in this section the part of the λProlog language that can be thought of as its core. Our presentation will be at two levels: we shall describe the logical underpinnings of the language and also attempt to describe it at the level of a usable programming language. Both aspects are required in later sections. The exposition at a logical level are needed to understand the semantics of the modules notion and to justify optimizations in its implementation. The presentation of the programming language is necessary to understand the value of modules as a pragmatic structuring construct.

1.1 Syntax

The logical language that underlies λProlog is ultimately derived from Church's simple theory of types [1]. This language is *typed* in the sense that every well-formed expression in it has a type associated with it. The language of types that is actually used permits a form of polymorphism. The type expressions are obtained from a set of *sorts*, a set of *type variables* and a set of *type constructors*, each of which is specified with a unique arity. The rules for constructing types are the following: (i) each sort and type variable is a type, (ii) if c is an n-ary type constructor and $t_1, \ldots, t_n$ are types, then $(c\ t_1\ \ldots\ t_n)$ is a type, and (iii) if α and β are types then so is $\alpha \rightarrow \beta$. Types formed by using (iii) are called *function* types. In writing function types, parentheses can be omitted by assuming that $\rightarrow$ is right associative. Type variables have a largely abbreviatory status in the language: they can appear in the types associated with expressions, but at a conceptual level such expressions can be used in a computation only after all the type variables appearing in them have been instantiated by closed types. A type is closed if it contains no type variables. However, these variables permit a succinct presentation of predicate definitions and, as we mention later, their instantiations at run-time can often be delayed. Thus, type variables provide a sense of polymorphism in λProlog.

At the level of concrete syntax, type variables are denoted by names that begin with an uppercase letter. The set of sorts initially contains only o, the boolean type, and *int*, the type of integers, and no type constructors are assumed. The user can define type constructors by using declarations of the form

$$kind\quad c\qquad type \rightarrow \ldots \rightarrow type.$$

The arity of the constructor c that is thus declared is one less than the number of occurrences of *type* in the declaration. Noting that a sort might be viewed as a nullary type constructor, a declaration of the above kind may also be used to add new sorts. As specific examples, the declarations

$$kind\quad i\qquad type.$$
$$kind\quad list\qquad type \rightarrow type.$$

add i to the set of sorts and define *list* as a unary constructor. The latter will be used below as a means for constructing types corresponding to lists of objects of a homogeneous type .

The *terms* of the language are constructed from given sets of constant and variable symbols, each of which is assumed to be specified with a type. The constants are categorized as the *logical* and the *nonlogical* ones. The logical constants consist of the following:

true	of type o, denoting the true proposition,
$\wedge$	of type $o \rightarrow o \rightarrow o$, representing conjunction.
$\vee$	of type $o \rightarrow o \rightarrow o$, representing disjunction,
$\supset$	of type $o \rightarrow o \rightarrow o$, representing implication,
sigma	of type $(A \rightarrow o) \rightarrow o$, representing existential quantification,
pi	of type $(A \rightarrow o) \rightarrow o$, representing universal quantification.

The symbols *sigma* and *pi* have a polymorphic type associated with them. These symbols really correspond to a family of constants, each indexed by a choice of ground instantiation for τ and a similar interpretation is intended for other polymorphic symbols.

In the machine presentation of nonlogical constants and variables, conventions similar to those in Prolog are used: both variables and constants are represented by tokens formed out of sequences of alphanumeric characters or sequences of "sign" characters, and those tokens that begin with uppercase letters correspond to variables. The underlying logic requires a type to be associated with each of these tokens. Symbols that consist solely of numeric characters are assumed to have the type *int*. For other symbols, an association is achieved by declarations of the form

type *constant* *type-expression.*

Such a declaration identifies the type of *constant* with the corresponding type expression. As examples, the declarations

$$\begin{array}{lll} \textit{type} & \textit{nil} & (\textit{list } A). \\ \textit{type} & :: & A \rightarrow (\textit{list } A) \rightarrow (\textit{list } A). \end{array}$$

define the constants *nil* and :: that function as constructors for homogeneous lists. Types of constants and variables may also be indicated by writing them in juxtaposition and separated by a colon. Thus the notation $X : int$ corresponds to a variable X of type *int*.

The terms in our logical language are obtained from the constant and variable symbols by using the mechanisms of function abstraction and application. In particular (i) each constant and variable of type τ is a term of type τ, (ii) if x is a variable of type τ and t is a term of type τ', then $\lambda x t$ is a term of type $\tau \rightarrow \tau'$, and (iii) if t_1 is a term of type $(\tau_2 \rightarrow \tau_1)$ and t_2 is a term of type τ_2, then $(t_1\ t_2)$ is a term of type τ_1. A term obtained by virtue of (ii) is referred to as an ***abstraction*** whose bound variable is x and whose scope is t. Similarly a term obtained by (iii) is called the application of t_1 to t_2.

Several conventions are adopted towards enhancing readability. Parentheses are often omitted by assuming that application is left associative and that abstraction is right associative. The logical constants $\wedge$, $\vee$ and $\supset$ are written as right associative infix operators. It is often useful to extend this treatment to nonlogical constants, and a device is included in λProlog for declaring specific constants to be prefix, infix or postfix operators. For instance, the declaration

$$infix\ 150 \quad xfy \quad ::.$$

achieves the same effect that the declaration $op(150, xfy, ::)$ achieves in Prolog: it defines :: to be a right associative infix operator of precedence 150.

An important notion is that of a *positive* term which is a term in which the symbol $\supset$ does not appear. We define an *atomic formula* or *atom* to be a term of type o that has the structure $(P\ t_1\ \ldots\ t_n)$ where P, the *head* of the atom, is either a nonlogical constant or a variable and $t_1, \ldots, t_n$, the *arguments* of the atom, are positive terms. Such a formula is referred to as a *rigid* atom if its head is a nonlogical constant, and as a *flexible* atom otherwise. Using the symbol A

to denote arbitrary atoms and A_r to denote rigid atoms, the classes of G-, D- and E-formulas are identified as follows:

$$\begin{array}{rcl} G & ::= & true \mid A \mid (G_1 \wedge G_2) \mid (G_1 \vee G_2) \mid sigma\ (\lambda x G) \mid \\ & & pi\ (\lambda x G) \mid (E \supset G) \\ D & ::= & A_r \mid G \supset A_r \mid pi\ (\lambda x D) \mid (D_1 \wedge D_2) \\ E & ::= & D \mid sigma\ (\lambda x E) \end{array}$$

A curious aspect of these syntax rules is the use of the symbols *pi* and *sigma*. These symbols represent universal and existential quantification respectively. The quantifiers that are used in conventional presentations of logic play a dual role: in the expression $\forall x P$, the quantifier has the function of binding the variable x over the expression P in addition to that of making a predication of the result. In the logical language considered here, these roles are separated between the abstraction operation and appropriately chosen constants. Thus the expression $\forall x P$ is represented here by $(pi\ (\lambda x P))$. The former expression may be thought of as an abbreviation for the latter, and we use this convention at a metalinguistic level below. A similar observation applies to the symbol *sigma* and existential quantification.

The G- and D-formulas determine the *programs* and *queries* of λProlog. A program consists of a list of closed D-formulas each element of which is referred to as a *program clause*, and a query or goal is an closed G-formula[2]. In writing the program clauses in a program in λProlog, the universal quantifiers appearing at the front are left implicit. A similar observation applies to the existential quantifiers at the beginning of a query. There are some other conventions used in the machine presentation of programs. Abstraction is depicted by \, written as an infix operator. Thus, the expression $\lambda X(X :: nil)$ is represented by $X\backslash(X :: nil)$. The symbols ∧ and ∨ are denoted by , and ; as in Prolog. Implications appearing at the top-level in program clauses are written backwards with `:-` being used in place of ⊃, and the symbol ⊃ in goal formulas is written as `=>`. Finally, a program is depicted by writing a sequence of program clauses, each clause being terminated by a period. An example of the use of these conventions is provided by the following clauses defining the familiar *append* predicate, assuming the types for *nil* and :: that were presented earlier.

$(append\ nil\ L\ L).$
$(append\ H :: L1\ L2\ H :: L3)$ `:-` $(append\ L1\ L2\ L3).$

Notice that not all the needed type information has been presented in these clauses: the types of the variables and of *append* have been omitted. These types could be provided by using the devices explained earlier. However, type declarations can be avoided in several situations since the desired types can be reconstructed [15]. For example, the type of *append* in the above program can be determined to be $(list\ A) \rightarrow (list\ A) \rightarrow (list\ A) \rightarrow o$. The type reconstruction algorithm that is used is sensitive to the set of clauses contained in the program. For example, if the program above included the clause

[2]This definition is more general than the one usually employed in that existential quantification is permitted over D formulas appearing to the left of implications in goals. This feature does not add anything new at a logical level, but is pragmatically useful as we see later. This extended definition is also used in [4].

(*append* (1 :: *nil*) (2 :: *nil*) (1 :: 2 :: *nil*)).

as well, then the type determined for *append* would be $(list\ int) \rightarrow (list\ int) \rightarrow (list\ int) \rightarrow o$ instead.

The example above shows the similarity of λProlog syntax to that of Prolog. The main difference is a curried notation, which is convenient given the higher-order nature of the language. There are similarities in the semantics as well as we discuss below.

1.1 Answering Queries from Programs

We present an operational semantics for λProlog by providing rules for solving a query in the context of a given program. The rules depend on the top-level logical symbol in the query and have the effect of producing a new query and a new program. Thus, the operational semantics induces a notion of computational state given by a program and a query. We employ structures of the form $\mathcal{P} \longrightarrow G$ where $\mathcal{P}$ is a listing of closed program clauses and G is a closed G-formula to represent such a state. We refer to these structures as *sequents*, and the idea of solving a query from a set of closed program clauses corresponds to that of constructing a derivation for an appropriate sequent.

Several auxiliary notions are needed in presenting the rules for constructing derivations. One of these is the notion of equality assumed in our language. Two terms are considered equal if they can be made identical using the rules of λ-conversion. We assume a familiarity on the part of the reader with a presentation of these rules such as that found in [5]. We need a substitution operation on formulas. Formally, we think of a substitution as a finite set of pairs of the form $\langle x, t\rangle$ where x is a variable and t is a term whose type is identical to that of x; the substitution is said to be closed if the second component of each pair in it is closed. Given a substitution $\{\langle x_i, t_i\rangle | 1 \leq i \leq n\}$, we write $F[t_1/x_1, \ldots, t_n/x_n]$ to denote the application of this substitution to F. Such an application must be done carefully to avoid the usual capture problems. The needed qualifications can be captured succinctly by using the λ-conversion rules: $F[t_1/x_1, \ldots, t_n/x_n]$ is equal to the term $((\lambda x_1 \ldots \lambda x_n F)\ t_1\ \ldots\ t_n)$. We also need to talk about *type instances* of terms. These are obtained by making substitutions for type variables that appear in the term. Finally, we are particularly interested in terms that do not have any type variables in them and we call such terms *type variable free.*

The various notions described above are used in defining the idea of an instance of a program clause.

Definition 1 *An instance of a closed program clause D is given as follows:*

(i) If D is of the form A_r or $G \supset A_r$, then any type variable free type instance of D is an instance of D.

(ii) If D is of the form $D_1 \wedge D_2$ then an instance of D_1 or of D_2 is an instance of D.

(iii) If D is of the form $\forall x D_1$, then an instance of $D_1[t/x]$ for any closed positive term t of the same type as x is an instance of D.

The restriction to (closed) positive terms forces an instance of a program clause to itself be a program clause. In fact, instances of program clauses have a very simple structure: they are all of the form A_r or $G \supset A_r$.

In describing the derivation rules, and thus the operational semantics of our language, we restrict our attention to type variable free queries. We present a more general notion of computation later based on this restricted definition of derivation.

Definition 2 *Let G be a type variable free query and let $\mathcal{P}$ be a program. Then a derivation is constructed for $\mathcal{P} \longrightarrow G$ by using one of the following rules:*

SUCCESS *By noting the G is equal to an instance of a program clause in $\mathcal{P}$.*

BACKCHAIN *By picking an instance of a program clause in $\mathcal{P}$ of the form $G_1 \supset G$ and constructing a derivation for $\mathcal{P} \longrightarrow G_1$.*

AND *If G is equal to $G_1 \wedge G_2$, by constructing derivations for the sequents $\mathcal{P} \longrightarrow G_1$ and $\mathcal{P} \longrightarrow G_2$.*

OR *If G is equal to $G_1 \vee G_2$, by constructing a derivation for either $\mathcal{P} \longrightarrow G_1$ or $\mathcal{P} \longrightarrow G_2$.*

INSTANCE *If G is equal to $\exists x G_1$, by constructing a derivation for the sequent $\mathcal{P} \longrightarrow G_1[t/x]$, where t is a closed positive term of the same type as x.*

GENERIC *If G is equal to $\forall x G_1$, by constructing a derivation for the sequent $\mathcal{P} \longrightarrow G_1[c/x]$, where c is a nonlogical constant of the same type as x that does not appear in $\forall x G$ or in the formulas in $\mathcal{P}$.*

AUGMENT *If G is equal to $(\exists x_1 \ldots \exists x_n D) \supset G$, by constructing a derivation for the sequent $D[c_1/x_1, \ldots, c_n/x_n], \mathcal{P} \longrightarrow G$, where, for $1 \leq i \leq n$, c_i is a nonlogical constant of the same type as x_i that does not appear in $(\exists x_1 \ldots \exists x_n D) \supset G$ or in the formulas in $\mathcal{P}$.*

To understand the operational semantics induced by these rules, let us assume a program given by the following clauses

```
(rev L1 L2) :-
      (((rev_aux nil L2),
      (pi (X\(pi (L1\(pi (L2\
            ((rev_aux X :: L1 L2) :- (rev_aux L1 X :: L2)))))))))
            => (rev_aux L1 nil)).
```

and consider solving the query $(rev\ 1 :: 2 :: nil\ \ 2 :: 1 :: nil)$. The first rule that must be used in a derivation is BACKCHAIN. Using it reduces the problem to that of solving the query

```
((rev_aux nil 2 :: 1 :: nil),
(pi (X\(pi (L1\(pi (L2\
        ((rev_aux X :: L1 L2) :- (rev_aux L1 X :: L2)))))))))
=> (rev_aux 1 :: 2 :: nil nil)
```

from the same program. The AUGMENT rule is now applicable and using it essentially causes the program to be enhanced with the clauses

```
(rev_aux nil 2 :: 1 :: nil).
(rev_aux X :: L1 L2) :- (rev_aux L1 X :: L2).
```

prior to solving the query (*rev_aux* 1 :: 2 :: *nil nil*). Using the BACKCHAIN rule twice in conjunction with the last clause produces the goal (*rev_aux nil* 2 :: 1 :: *nil*). The derivation attempt now succeeds because the goal is an instance of program clause.

The above example indicates the programming interpretation given to logical formulas and symbols by the operational semantics. Program clauses of the form $\forall x_1 \ldots \forall x_n A_r$ and $\forall x_1 \ldots \forall x_n (G \supset A_r)$ function in a sense as procedure definitions: the head of A_r represents the name of the procedure and, in the latter case, the body of the clause, G, corresponds to the body of the procedure. From an operational perspective, every program clause is equivalent to a conjunction of clauses in this special form, and a program is equivalent to a list of such clauses. Thus both correspond to a collection of procedure definitions. Goals correspond to search requests with the logical symbols appearing in them functioning as primitives for specifying the search structure. Thus, in searching for a derivation, $\wedge$ gives rise to an AND branch, $\vee$ to an OR branch and *sigma* to an OR branch parameterized by a substitution. These symbols are used in a similar fashion in Prolog. The symbols $\supset$ and *pi*, on the other hand, do not appear in Prolog goals. The treatment of these symbols is interesting from a programming viewpoint. The first symbol has the effect of augmenting an existing program for a limited part of the computation. Thus, this symbol corresponds to a primitive for giving program clauses a scope. The symbol *pi* similarly corresponds to a primitive for giving names a scope; processing this symbol requires a new name to be introduced for a portion of the search. A closer look at the operational semantics reveals a similarity between the interpretation of *pi* and the treatment given to existential quantifiers in E-formulas through the AUGMENT rule. This is not very surprising since the formulas $\forall x(D(x) \supset G)$ and $(\exists x D(x)) \supset G$ are equivalent in most logical contexts, assuming x does not appear free in G. From a pragmatic perspective, then, the existential quantifier in E-formulas enables a name to be made local to a set of procedure definitions, *i.e.*, it provides a means for information hiding.

A computation in λProlog corresponds to constructing a derivation for a query from a given program. We are generally interested in extracting a value from a computation. In the present context, this can be made clear as follows.

Definition 3 *Let $\mathcal{P}$ be a collection of program clauses and let G be a type variable free query of the form $\exists x_1 \ldots \exists x_n G_1$; the variables $x_1, \ldots, x_n$ are assumed to be implicitly quantified here. An answer to G in the context of $\mathcal{P}$ is a closed substitution $\{\langle x_i, t_i\rangle \mid 1 \le i \le n\}$ such that $\mathcal{P} \longrightarrow G_1[t_1/x_1, \ldots, t_n/x_n]$ has a derivation.*

In general our queries may have type variables in them. The answers to such a query are given by the answers to each of its type variable free type instances.

Our ultimate interest is in a procedure for carrying out computations of the kind described above and for extracting results from these. The rules for constructing derivations provide a structure for such a procedure but additional mechanisms are needed. One problem involves instantiations for type variables. There is usually insufficient information for choosing instantiations for these at the points indicated. This problem can be overcome by allowing type variables into the computation and by using unification to incrementally determine their instantiations. A similar problem arises with existential quantifiers in queries. For example, solving a query of the form $\exists x G$ requires a closed term t to be produced that makes $G[t/x]$ solvable. The usual mechanism employed in these cases is to replace x with a *logic* variable, *i.e.*, a place-holder, and to let an appropriate instantiation be determined by unification. However, this mechanism must be used with care in the present situation. First, the unification procedure that is used must incorporate our enriched notion of equality, *i.e.*, higher-order unification [6] must be used. Second, the treatment of universal quantifiers requires unification to respect certain constraints. For example, consider the query $\exists x \forall y p(x, y)$, where p is a predicate constant. Using the mechanisms outlined, this query will be transformed into $p(X, c)$, where c is a new constant and X is a logic variable. Notice, however, that X must not be instantiated with a term that contains c in it. A solution to this problem is to add a numeric tag to every constant and variable and to use these tags in constraining the unification process [3, 18].

A suitable abstract interpreter can be developed for λProlog based on the above ideas[3]. In actually implementing this interpreter, two additional questions arise. First, there is some nondeterminism involved: in solving an atomic goal, a choice has to be made between program clauses and in solving $G_1 \vee G_2$ a decision has to be made between solving G_1 and G_2. The usual device employed here is to use a depth-first search with backtracking. The second question concerns the implementation of implications in queries. To understand the various problems that arise here, let us consider a query of the form $(D \supset G_1) \wedge G_2$. This query results in the query $D \supset G_1$ which must be solved by adding (the clauses in) D to the program, solving the clauses in G_1 and then removing D. The addition of code follows a stack based discipline and can be implemented as such. However, if a compilation model is used, some effort is involved in spelling out a scheme for achieving the addition and deletion of code. Moreover the "program clauses" that are added might now contain logic variables in them. Thus, consider solving the goal $\exists L(rev\ 1 :: 2 :: nil\ L)$ using the clause for *rev* presented earlier in this section. The program would at a certain stage have to be augmented with the clause $(rev_aux\ nil\ L)$ where L is now a logic variable. In general, we need now to think of procedures as blocks of code *and* bindings for some variables. Continuing now with the solution of the query $(D \supset G_1) \wedge G_2$, the goal G_2 will be attempted after the first conjunct is solved. A failure in solving this goal might require an alternative solution to G_1 to be generated. Notice, however, that an attempt to find such a solution must be made in a context where the program once again contains D. An implementation of our language must support the needed context switching ability.

Implementation techniques have been devised for solving the various problems mentioned above

[3]Actually, the proper treatment of type variables in a computation is still an open issue. However, a discussion of this matter is orthogonal to our present purposes.

[7, 16, 17], resulting in an abstract machine and a compilation scheme for the core language described in this section. We do not discuss this explicitly here, and will rely on the reader's intuition and indulgence when alluding to these ideas later in the paper. However, the discussion of modules will require a closer acquaintance with the scheme used for implementing implications in queries, and we then supply some further details.

Before concluding this section, it is interesting to note the connection between our notion of computation and deduction in a logical context. The following proposition describes this connection.

Proposition 4 *Let $\mathcal{P}$ be a program and let $\mathcal{P}'$ be the collection of all the type variable free type instances of formulas in $\mathcal{P}$. Further, let G be a type variable free query. Then there is a derivation for $\mathcal{P} \longrightarrow G$ if and only if G follows from $\mathcal{P}'$ in intuitionistic logic.*

Only the *only if* part of this proposition is non-trivial. For the most part, this follows from the existence of uniform proofs for sequents of the kind we are interested in; see, *e.g.*, [13] and [18] for details. One additional point to note is the treatment of existential quantifiers in E-formulas. However, this causes no problem because the introduction of existential quantifiers in assumptions can always be made the last step in intuitionistic proofs.

4 Modules

The language described thus far only permits programs that are a monolithic collection of kind, type, and operator declarations together with a set of procedure definitions. Modules provide a means for structuring the space of declarations and also for tailoring the definitions of procedures depending on the context. The ultimate purpose of this feature is to allow programs to be built up from logical segments which are in some sense separate.

At the very lowest level, the module feature allows a name to be associated with a collection of declarations and program clauses. An example of the use of this construct is provided by the following sequence of declarations that in effect attaches the name *lists* with the list constructors and some basic list-handling predicates:

```
module      lists.
infix 150   ::      xfy.
kind  list  type -> type.
type  nil   (list A).
type  ::    A -> (list A) -> (list A).

(append nil L L).
(append (H :: L1) L2 (H :: L3)) :- (append L1 L2 L3).

(member H (H :: L)).
(member X (H :: L)) :- (member X L).

(length 0 nil).
(length N (H :: L) :- ((length N1 L).N is N1 + 1).
```

One way to think of this module declaration is as a declaration of a list "data type". This data type can be made available in specific contexts by using the name *lists* in a manner that we describe presently. This discussion will bring out the intended purpose of the modules feature. However, there is one use that can already be noted. Looking at the *lists* module above, we see that the types of the predicates defined in it have not been provided. These types can be reconstructed, but, as we noted in Section 3, the types "inferred" depend on the set of available program clauses. The module boundary provides a notion of scope that is relevant to this reconstruction process: looked at differently, the types of all the symbols appearing in the clauses in a module are completely determined once the module is parsed.

The meaning of the module feature is brought out by considering its use in programming. In the presence of modules, we enhance our goals to include a new kind of expression called a *module implication*. These are expressions of the form M ==> G, where M is a module name. Goals of the new sort have the intuitive effect of adding M to the program before solving G. In making this precise, however, the effect of M on two different components have to be made clear: on the type, kind and operator declarations and on the procedure definitions.

The effect on the space of declarations that we assume here is simple. All the associations present in M become available on adding M to the context. This is really a *static* effect in that it provides a context in which to parse the goal G in a larger goal M ==> G. As a concrete example, consider the goal

lists ==> (*append* 1 :: 2 :: *nil* 3 :: *nil* L).

In parsing this query, there is a need to determine the types of *append* and of ::. The semantics attributed to the modules feature requires the types associated with these tokens in the module *lists* to be assumed for this purpose. This appears to be the most natural course, given that we expect the definition of *append* provided in *lists* to be useful in solving this query.

From the perspective of procedure definitions, we assume the semantics for modules that is presented in [11]. Within this framework, the dynamic aspects of the module feature are explained by a translation into the core language. Thus, a module is thought of as the conjunction of the program clauses appearing in it. For instance, the *lists* module corresponds to the conjunction of the clauses for *append*, *member* and *length*. Under this interpretation, a module corresponds to a D-formula as described in the last section. Now if module M corresponds to the formula D, the query M ==> G is thought of as the goal D => G. The run-time treatment of module implication is then determined by the AUGMENT rule presented in the last section. In particular, solving the goal M ==> G calls for solving the goal G after adding the predicate definitions in the module M to the existing program.

The analogy between a module and a data type raises the question of whether some aspects of an implementation might be hidden. Our language permits constant names to be made local to a module, thus allowing for the hiding of a data structure. To achieve this effect, a declaration of the form

local *constant*, ..., *constant*.

can be placed within a module. The names of the constants listed then become unavailable outside the module. For example, adding the declaration

local ::.

to the *lists* module has the effect of hiding the list constructor ::.

The static effect of the local construct is easy to understand: only some names are available when the module is added to a context. From a dynamic perspective, another issue arises. Can constants defined to be local eventually become visible outside through computed answers? The expectation is that they should not become so visible. This effect can be achieved by thinking of local constants really as variables quantified existentially over the scope of the conjunction of program clauses in the module. As an example, consider the following module

```
module     store.
local  emp, stk.
kind   store  type -> type.
type   emp    (store A).
type   stk    A -> (store A) -> (store A).
initialize emp.
(enter X S (stk X S)).
(remove X (stk X S) S).
```

This module implements a *store* data type with initializing, adding and removing operations. At a level of detail, the store is implemented as a stack. However, the intention of the local declarations is to hide the actual representation of the store. Now, from the perspective of dynamic effects, the module corresponds to the formula

```
∃Emp∃Stk(
        (initialize Emp) ,
        (pi (X\(pi (S\(enter X S (Stk X S)))))),
        (pi (X\(pi (S\(remove X (Stk X S) S)))))).
```

This formula has the structure of an E-formula and in fact every module corresponds in the sense explained to an E-formula. Referring to this formula as *EStore*, let us consider solving a goal of the form $\exists X(Store$ ==> $G(X))$. The semantics of this goal requires solving the goal $\exists X(EStore$ => $G(X))$. Under the usual treatment of existential quantifiers, this results in the goal $(EStore$ => $G(X))$ where X is now a logic variable. Using the AUGMENT rule, this goal is solved by instantiating the existential quantifiers at the front of *EStore*, adding the resulting D-formula to the program and then solving $G(X)$. The important point to note now is that any substitution that is considered for X must not have the constants supplied for *Emp* and *Stk* appearing in it. Thus the semantics attributed to modules and local declarations achieves the intended dynamic effect.

While module implication is useful for making modules available at the top-level, modules may themselves need to interact. For instance, a module that contains sorting predicates might need the declarations and procedure definitions in the *lists* module and a module that implements graph-search might similarly need the *store* and *lists* modules. The needed interaction is obtained in λProlog by placing an *import* declaration in the module which needs other modules. The format of such a declaration is the following:

import $M1, \ldots, Mk$.

In a declaration of this sort, $M1, \ldots, Mk$ must be names of other modules that are referred to as the *imported* modules. A declaration of this sort has, once again, a static and a dynamic effect on the module in which it is placed, *i.e.*, the *importing* module. The static effect is to make all the declarations in the imported modules, save those hidden by local declarations, available in the importing module. These declarations can be used in parsing the importing module and also become part of the declarations provided by that module. The intended dynamic effect, on the other hand, is to make the procedure definitions in the imported modules available for solving the goals in the bodies of program clauses that appear in the importing module. This effect can actually be explained by using module implication [11]. Let us assume that the clause P :- G appears in a module that imports the modules $M1, \ldots, Mk$. The dynamic semantics involves interpreting this clause as the following one instead:

P :- ($M1$ ==> $\cdots$ (Mk ==> G)).

Observe that using this clause involves solving the goal ($M1$ ==> $\cdots$ (Mk ==> G)) that ultimately causes the program to be enhanced with the clauses in $M1, \ldots, Mk$ before solving G.

The definition of the module *graph_search* presented in Figure 1 illustrates the usefulness of the module interaction facility provided by *import*. The definitions of the predicates *start_state*, *final_state*, *soln* and *expand_node* have not been presented here, but we anticipate the reader can supply these. The important aspect to note is the use that is made of the declarations and procedure definitions in the modules *lists* and *store*. For example, the type

$(list\ A) \rightarrow (store\ A) \rightarrow (list\ A) \rightarrow (store\ A) \rightarrow o$

will be reconstructed for *add_states*. This type uses type constructors defined in in the modules *lists* and *store*. Similarly, the procedure *member* defined in *lists* and the procedures *initialize*, *enter* and *remove* defined in *store* are used in the program clauses in the module *graph_search*. A particularly interesting aspect is the interaction between the modules *graph_search* and *store*. Notice that the "constants" *emp* and *stk* used in *store* are not visible in *graph_search* and cannot be used explicitly in the procedures appearing there. Thus, importing *store* provides an abstract notion of a store without opening up the actual implementation. For example, the current stack-based realization of the store can be replaced by a queue-based one without any need to change the *graph_search* module so long as the operations *initialize*, *enter* and *remove* are still supported. This change will have an effect on the behavior of *g_search* though, changing it to a procedure that conducts breadth-first search as opposed to the current depth-first search.

The pragmatic utility of the module feature and of the scoping ability provided by the new logical symbols in our language is an important issue to consider and detailed discussions of this aspect appear in [10] and [11]. Our interest in this paper is largely on implementation issues, especially those arising out of the module notion. From this perspective, it is necessary to understand carefully the dynamic interactions that can arise between modules through the use of the *import* statement. We therefore present an example that illustrates some of these interactions. Figure 2 contains a collection of interacting modules and Figure 3 exhibits the process of solving the query ($m1$ ==> (p X)) given these definitions. In presenting this solution attempt, we use a linear format based

```
module graph_search.
import lists, store.

(g_search Soln) :-
        ((init_open Open), (expand_graph Open nil Soln)).

(init_open Open) :-
        ((start_state State), (initialize Op), (enter State Op Open)).

(expand_graph Open Closed Soln) :-
        (remove State Open ROp),
        (((final_state State), (soln State Soln));
        ((expand_node State NStates),
         (add_states NStates ROp (State :: Closed) NOp),
         (expand_graph NOp (State :: Closed) Soln))).

(add_states nil Open Closed Open).
(add_states (St :: RSts) Open Closed NOpen) :-
        ((member St Closed), (add_states RSts Open Closed NOpen)).
(add_states (St :: RSts) Open Closed NOpen) :-
        ((enter St Open NOp), (add_states RSts NOp Closed NOpen)).

        ...
```

Figure 1: A Module Implementing Graph Search

```
module m1.
import m2.
(p X) :- (q X),(t X).
(t b).
(s X) :- (r X).
```

```
module m2.
import m3.
(p b).
(q X) :- (s X).
```

```
module m3.
type r i -> o.
(r a).
(r b).
```

Figure 2: A Set of Interacting Modules

$m1$	?- (pX)		
$m2, m1$	?- $(q\ X)$		
$m3, m2, m1$	?- $(s\ X)$		
$m2, m3, m2, m1$	?- $(r\ X)$	X <- a	$SUCC$
$m2, m1$	?- $(t\ a)$	$FAIL$	
$m2, m3, m2, m1$	?- $(r\ X)$	X <- b	$SUCC$
$m2, m1$	?- $(t\ b)$	$SUCC$	

Figure 3: Solving $(m1$ ==> $(p\ X))$ Given the Modules in Figure 2

on the notion of derivation presented in Section 3 but augmented with the use of logic variables. Further, we use lines of the following form

$M1, \ldots, Mn$?- $G(X)$

where $G(X)$ is an atomic goal and $M1, \ldots, Mn$ are module names. Such a line indicates that $G(X)$ is to be solved from a program given by the collection of clauses in $M1, \ldots, Mn$. We refer to this list of modules as a program context. Now, the attempt to solve this goal proceeds by trying to match the goal with the head of some clause. If this attempt is successful, the line is annotated by a binding for the logic variables, *e.g.*, by an expression such as X <- a. In the case that the match results in additional goals, the following lines pertain to the solution of these goals. If no match is possible or if the match results directly in a success, the line is further annotated with the word $FAIL$ or $SUCC$. In the former case, the succeeding lines indicate the solution attempt after backtracking and in the latter case they indicate an attempt to solve the remaining goals.

Let us consider now the attempt to solve the mentioned goal, $(m2$ ==> $((p\ X))$. The initial program context is empty, but dealing with the module implication causes $m1$ to be added to it. The goal to be solved now is $(p\ X)$. There is only one clause available for p and this is interpreted as if it were

$(p\ X)$:- $(m2$ ==> $((q\ X),(t\ X)))$

since $m1$ imports $m2$. Module $m2$ is therefore added to the program context and the goal to be solved reduces to $(q\ X),(t\ X)$. Although not relevant to the solution of the present goal, notice

that module $m2$ also contains a clause for p. The new program context thus contains an enhanced definition for this predicate and an implementation must be capable of combining code from different sources to produce the desired effect. Tracing through the solution attempt a few more steps, we see that the use of the second clause in module $m1$ results in an attempt to solve $(r\ X)$. There are two clauses for this predicate in the relevant program context and these are used in order. Note that this interaction could not have been predicted from the static structure of $m1$ alone: there is no compile-time indication that code in module $m3$ might be used in solving goals appearing in the bodies of clauses in $m1$. A compilation scheme must therefore be sensitive to the fact that the definition of procedures used within modules are determined dynamically. Continuing with the solution attempt, $(r\ X)$ is solved successfully with X being bound to the constant a. The task now becomes one of solving the goal $(t\ a)$. Notice that the program context for this goal includes only $m1$ and $m2$, *i.e.*, an implementation must support this kind of context switching. When this goal fails, backtracking now requires an alternative solution to $(r\ X)$ to be found. However, this solution attempt must take place in a resurrected context, as indicated in the figure. Once again, an implementation of the module feature must be capable of supporting this kind of reinstatement of earlier contexts.

We consider in the next section the various implementation issues pertaining to the dynamic behavior of modules that are raised by the above example. We note that a desirable feature of an implementation scheme is that it should permit a separate compilation of each module; this is in some sense indicative of the ability of this feature to split up a program into logically separate parts. The scheme that we present for implementing the dynamic behavior exhibits this facet — separate segments of compiled code are produced for each module and these are linked together dynamically to produce a desired program context. However, the idea of separate compilation is somewhat more problematic at the level of static interaction. The main issue is that the parsing of an importing module requires the various type, kind and operator declarations in the imported modules, implying a dependence in compilation. This kind of behavior is, however, not unique to our context. The usual solution to this problem is to introduce the idea of an interface between modules. Specialized to our context, this involves assigning a set of declarations to a module name. This assignment may act in a prescriptive fashion on the actual set of declarations appearing in the module in the sense that they may be required to conform to the "interface" requirements. With regard to importation, on the other hand, the interface declarations could control what is visible. One consequence of this view is that the association of types with constants might be hidden. Such an occlusion must be accompanied with a hiding of the constant itself and thus affects the dynamic behavior. However, this behavior can be modelled by the use of implicit local declarations[4]. A proper use of this idea will require predicate definitions also to be hidden. This ability is not supported within the current language: the ability to quantify existentially over predicate names requires an extension of the syntax of D-formulas. The extension in syntax can be easily accomplished as indicated in [4] and [10]. Although we do not treat this matter explicitly here, the desired extension does not cause any semantical problems and, as indicated in [16], can also be accommodated within our

[4] A related proposal is contained in [12]. However, the suggestion there is to determine the local declarations dynamically, depending on the goal to be solved. This appears not to help with the "static" problem discussed here and also makes it difficult to generate code for a module independent of its use.

implementation scheme.

While the use of an interface as a method for prescribing interactions in this manner has several interesting aspects, a more conservative view of it is also possible. The interface declarations may be viewed simply as a distillate of the compilation of the module in question. Regardless of which view is taken, we assume here that, when a module is being compiled, all the type, kind and operator declarations obtainable from the imported modules are known. The scheme that we present in the next section then generates the code for capturing the dynamic behavior of a module by using only these interfaces and parsing the module in question. In this sense, our scheme is capable of supporting the idea of separate compilation.

5 Implementing the Dynamic Semantics of Modules

The crucial issue that must be dealt with in an implementation of the dynamic aspects of modules is the treatment of module implication. In particular, we are interested in the compilation of goals of the form M ==> G. Within a model that supports separate compilation, the production of code from the predicate definitions appearing in M must be performed independently of this goal. The compiled effect of this goal must then be to enhance the program context by adding the code in M to it. Under this view, the symbol ==> becomes a primitive for linking code. The crucial issues within an implementation thus become those of what structures are needed for realizing this linking function and of what must be produced as a result of the compilation of a module to facilitate the linking process at run-time.

We have developed a scheme elsewhere [16] for implementing goals that contain implications. The dynamic semantics of module implication coupled with some features of the mentioned scheme make it an apt one to adapt to the present context. The essence of our scheme is to view a program as a composite of compiled code and a layered access function to this code. The execution of an implication goal causes a new layer to be added to an existing access function. Thus, consider an implication goal of the form $(C_1, \ldots, C_n)$ => G where, for $i \leq i \leq n$, C_i is a closed program clause of the form $\forall x_1 \ldots \forall x_n A_r$ or $\forall x_1 \ldots \forall x_n (G \supset A_r)$[5]. Each C_i corresponds to a partial definition of a procedure that must be added to the front of the program while an attempt is made to solve G. These clauses can be treated as an independent program segment and compiled in a manner similar to that employed in the WAM. Let us suppose that the clauses define the predicates $p_1, \ldots, p_r$. The compilation process then results in a segment of code with r entry points, each indexed with the name of a predicate. In our context, compilation must also produce a procedure that we call *find_code* that performs the following function: given a predicate name, this procedure returns the appropriate entry point in the code segment if the name is one of $p_1, \ldots, p_r$ and an indication of

[5]In the general case, every implication goal can be transformed into one of the form

$$Q_1 X_1 \ldots Q_m X_m((C_1(X_1, \ldots, X_m), \ldots, C_n(X_1, \ldots, X_m)) \texttt{=>} G(X_1, \ldots, X_m))$$

where Q_i is $\exists$ or $\forall$ and $C_i(x_1, \ldots, x_m)$ is a program clause of the sort indicated but which may depend on the variables $x_1, \ldots, x_m$. Existential quantifiers may arise in considerations of module implication only if the module notion is enriched to allow for parameterization. Universal quantifiers do arise indirectly through *local* declarations whose treatment is considered later in this section.

failure otherwise. This function can be implemented in several different ways such as through the use of a hash-function, but the details will not concern us here. Returning now to the implication goal, its execution results in a new access function that behaves as follows. Given a predicate name, *find_code* is invoked with it. If this function succeeds, then the code location that it produces is the desired result. Otherwise the code location is determined by using the access function in existence earlier.

The process of enhancing a context described above is incomplete in one respect: the new clauses provided for $p_1, \ldots, p_r$ may in fact be adding to earlier existing definitions for these predicates. To deal with this situation, the compilation process must produce code for each of these predicates that does not fail eventually, but instead looks for code for the relevant predicate using the access function existing earlier. Rather than carrying out this task each time it is needed, using an idea from [8], it can be done once at the time the new program context is set up. The idea used is the following. A vector of size r can be associated with the implication goal, with the ith entry in this vector corresponding to the predicate p_i. Now, the compilation of the body of the implication goal creates a procedure called *link_code* whose purpose is to fill in this vector when the implication goal is executed. This procedure essentially uses the name of each of the predicates and the earlier existing access function to compute an entry point to available code or, in case the predicate is previously undefined, to return the address of a failing procedure. To complement the creation of this table, the last instruction in the code generated for each of the predicates p_i must actually result in a transfer to the location indicated by the appropriate table entry.

In the framework of a WAM-like implementation, the layered access function described above can be realized by using what are called *implication point records.* These records are allocated on the local stack and correspond essentially to layers in the access function. The components of such a record, based on the discussions thus far, are the following:

(i) the address of the *find_code* procedure corresponding to the antecedent of the implication goal,

(ii) a positive integer r indicating the number of predicates defined by the program clauses in the antecedent,

(iii) a pointer to an enclosing implication point record, and thereby to the previous layer in the access function, and

(iv) a vector of size r that indicates the next clause to try for each of the predicates defined in the antecedent of the implication goal.

The program context existing at a particular stage is indicated by a pointer to a relevant implication point record which is contained in a register called `I`. Now a goal such as $(C_1, \ldots, C_n)$ `=>` G is compiled into code of the form

push_impl_point t
  { Compiled code for G }
pop_impl_point

In this code, t is the address of a statically created table for the antecedent of the goal that indicates the address of its *find_code* and *link_code* procedures and the number of predicates defined. The *push_impl_point* instruction causes a new implication point record to be allocated. The first three components of this record are set in a straightforward manner using the table indicated and the contents of the I register. Filling in the last component involves running *link_code* using the access function provided by the I register. The final action of the instruction is to set the I register to point to the newly created implication point record. The effect of the *pop_impl_point* instruction is to reset the program context. This is achieved simply by setting the I register to the address of the enclosing implication point record, a value stored in the record the I register currently points to.

There are a few points about the scheme described that are worth mentioning. First, under this scheme the compilation of an atomic goal does not yield an instruction to transfer control to a particular code address. Rather, the instruction produced must use an existing access function (indicated by the I register) and an index generated from the name of the predicate to locate the relevant code. Notice that this behavior is to be anticipated, given the dynamic nature of procedure definitions. The second observation pertains to the resurrection of a context upon backtracking. Under our scheme, the program context is reduced to the contents of a single register. By saving these contents in a WAM-like choice point record and by retaining implication point records embedded within choice points, the necessary context switching can be easily achieved.

We turn finally to the implementation of module implication. Let us consider first the treatment of a module implication of the form M ==> G where M is a module with no local declarations and no import statements. From the perspective of dynamic semantics, M can be reduced to a conjunction of closed D-formulas of the form $\forall x_1 \ldots \forall x_n A_r$ or $\forall x_1 \ldots \forall x_n (G \supset A_r)$, *i.e.*, of the form just considered. Thus the scheme outlined above can be applied almost without change to the treatment of this kind of module implication. Under this scheme, the compilation of the module M must produce code that implements the relevant *find_code* and *link_code* procedures in addition to the compiled code for the various predicates defined. The linking operation corresponding to ==> effectively amounts to setting up an implication point record. The main task involved in this regard is that of executing the *link_code* function which in a sense links the predicate definitions in the module to those already existing in the program.

The handling of local declarations does not pose any major complications. The treatment of a goal of the form E => G that is indicated by the operational semantics essentially requires the existential quantifiers at the front of E to be replaced by new constants and the resulting D-formula to be added to the existing program. Implementing this idea results in the local constants in E being conceived of as constants but with a numeric tag that prevents them from appearing in terms substituted for logic variables in G. At a level of detail, these constants can be identified with cells in an implication point record and the *push_impl_point* instruction has the additional task of allocating these cells and of tagging them with the appropriate numeric value.

The only remaining issue is the treatment of import declarations. Let us assume that a module M imports the modules $M1$,$M2$ and $M3$. From the perspective of dynamic semantics, this importation has an effect largely on the clauses appearing in M. Let P :- G be one of these clauses. Based on the semantics of importing, this clause is to be interpreted as the clause

P :- ($M1$ ==> ($M2$ ==> ($M3$ ==> G))).

This translation actually indicates a straightforward method for implementing the effect of importation: the body of the clause can be compiled into the code generated for G nested within a sequence of *push_impl_point* and *pop_impl_point* instructions. Noting that module M may contain several clauses, an improvement is possible in this basic scheme. We identify with a module two additional functions that we call *load_imports* and *unload_imports*. In the case of module M, executing the first of these corresponds conceptually to executing the sequence

push_impl_point $M1$
push_impl_point $M2$
push_impl_point $M3$

and, similarly, executing the second corresponds to executing a sequence of three *pop_impl_point* instructions. The address of these two functions is included in the implication point record created when a module is added to the program context. From the perspective of compilation, the code that is generated for the clause considered now takes the following shape:

{Code for unifying the head of the clause }
push_import_point M
    {Compiled code for goal G}
pop_import_point M

The *push_import_point* instruction in this sequence has the effect of invoking the *load_imports* function corresponding to module M and the *pop_import_point* instruction similarly invokes the *unload_imports* function.

The scheme described above assumes that the address of the compiled code and the various functions associated with a module can be indexed by the name of the module. This information is organized into entries in a global table with each entry having the following components:

(i) r, the number of predicates defined in the module,

(ii) the starting address for the compiled code segment for the predicates defined in the module; *find_code* will return offsets from this address,

(iii) the address of the *find_code* routine for the module,

(iv) the address of the *link_code* routine for the module,

(v) the address of the *load_imports* routine for the module, and

(vi) the address of the *unload_imports* routine for the module.

In reality not every module is loaded into memory at the beginning of a program and hence not every module has an entry in the global table. If a module that does not already reside in memory is needed, then a loading process brings the various segments of code in and creates an appropriate entry in the global table for the module. It should be clear by this point that the codes and information needed for each module can be obtained by a compile-time analysis of that module and the necessary interface definitions.

6 Controlling redundancy in search

The semantics presented for module implication and for the *import* statement could result in the same module being added several times to a program context. This has a potential drawback: it may result in redundancy in the search for a solution to a goal and the same solution may also be produced several times. To understanding this possibility, let us consider the following definition of a module called *sets* which imports the module *lists* presented in Section 4.

> *module sets.*
> *import lists.*
> *type subset* (*list* A) $\rightarrow$ (*list* A) $\rightarrow$ *o*.
> *subset nil* L.
> (*subset* $X :: L1\ L2$) :- ((*member* $X\ L2$), (*subset* $L1\ L2$)).

Assume now that an attempt is made to solve the goal

> *sets* ==> *subset* $1 :: 2 :: 4 ::$ *nil* $1 :: 2 :: 3 ::$ *nil*.

Using the linear format described in Section 4, part of the effort in solving this goal is represented by the following sequence:

sets	?-	(*subset* $1 :: 2 :: 4 ::$ *nil* $1 :: 2 :: 3 ::$ *nil*)
lists, sets	?-	(*member* $1\ 1 :: 2 :: 3 ::$ *nil*) *SUCC*
lists, sets	?-	(*subset* $2 :: 4 ::$ *nil* $1 :: 2 :: 3 ::$ *nil*)
lists, lists, sets	?-	(*member* $2\ 1 :: 2 :: 3 ::$ *nil*)
lists, lists, sets	?-	(*member* $2\ 2 :: 3 ::$ *nil*) *SUCC*
lists, lists, sets	?-	(*subset* $4 ::$ *nil* $1 :: 2 :: 3 ::$ *nil*)
lists, lists, lists, sets	?-	(*member* $4\ 1 :: 2 :: 3 ::$ *nil*)

It is easily seen that the attempt to solve the last goal in this sequence in the indicated program context will fail. Notice however, that a considerable amount of redundant search will be performed before this decision is reached: there are three copies of the module *lists* in the program context and the clauses for *member* in each of these will be used in turn in the solution attempt. A similar redundancy is manifest in the answers that are computed under the semantics provided. For instance, the query

> *sets* ==> *subset* $S\ 1 :: 2 ::$ *nil*

will result in the substitution $1 :: 2 ::$ *nil* for S being generated twice through the use of the clauses in two different copies of the module *lists*.

The extra copies of the module *lists*, while leading to redundancy in search, do not result in an ability to derive new goals or to find additional answers. Adding these copies also results in a runtime overhead: given the implementation scheme of the previous section, the addition of each copy results in the creation of an implication point record, thereby consuming both space and time. A pragmatic question to ask, therefore, is whether the number of copies of any module in

a program context can be restricted to just one. In answering this question there is an important principle to adhere to. It is desirable that the *logical semantics* of our language not be altered. In particular, we still want to be able to understand our language by using the derivation rules presented in Section 3 and to understand the dynamic semantics of modules through the devices discussed in Section 4. This principle is important because, as argued in [12], several interesting tools for analyzing the behavior of programs depend on this kind of a logical understanding of programming language constructs. In light of this principle, the question raised can be changed to one of the following sort: is it possible to preserve the important *observable* aspects of the given semantics while perhaps changing the details of the operational semantics so as to produce a preferred computational behavior. An affirmative answer to this question permits us to have the best of both worlds. The original semantics can be used for analyzing the interesting aspects of the behavior of programs while an actual implementation can be based on a modified set of derivation rules.

In the context being considered, the important aspects of program behavior are the set of queries that can be solved and the answers that can be found to any given query. Both aspects are completely determined by the set of sequents that have derivations. Thus, based on the above discussion, we might contemplate changing the underlying derivation rules for our language so as to reduce the *number* of derivations for any sequent while preserving the set of sequents that have derivations. With this in mind, we observe that the main source of redundancy in the example considered above is the $AUGMENT$ rule. Assume that we want to solve a goal of the form D => G. The $AUGMENT$ rule requires D to be added to the program context before attempting to solve G. Notice, however, that if D is already available in the program context, this addition is not likely to make a derivation of G possible where it earlier was not. A more interesting case is when the implication goal is of the form $(\exists x_1 \ldots \exists x_n D)$ => G. In this case the AUGMENT rule requires the addition of $D[c_1/x_1, \ldots, c_n/x_n]$ (for a suitable choice of c_is) to the program prior to the attempt to solve G. However, if the program already contains a clause of the form $D[c'_1/x_1, \ldots, c'_n/x_n]$, the addition is again redundant from the perspective of being able to solve G.

In the rest of the section we prove the observations contained in the previous paragraph. Towards this end, we define an alternative to the AUGMENT rule.

Definition 5 *Let G be a type variable free query and let $\mathcal{P}$ be a program. Then the AUGMENT′ rule is applicable if G is of the form $(\exists x_1 \ldots \exists x_n D)$ => G' and can be used to construct a derivation for $\mathcal{P} \longrightarrow G$ as follows:*

(i) If a formula of the form $D[c'_1/x_1, \ldots, c'_n/x_n]$ does not appear in $\mathcal{P}$, then by constructing a derivation for $D[c_1/x_1, \ldots, c_n/x_n], \mathcal{P} \longrightarrow G'$ where, for $1 \leq i \leq n$, c_i is a nonlogical constant of the same type as x_i not appearing in the formulas in $\mathcal{P}, G$.

(ii) If a formula of the form $D[c'_1/x_1, \ldots, c'_n/x_n]$ appears in $\mathcal{P}$, then by constructing a derivation for $\mathcal{P} \longrightarrow G'$.

Let us refer to the derivation rules presented earlier as $DS1$ and let $DS2$ be obtained from $DS1$ by replacing AUGMENT with AUGMENT′. We say that a sequent has a derivation in $DS1$ ($DS2$)

if a derivation can be constructed for it by using the rules in $DS1$ (respectively, $DS2$). We now make the following observation about derivations in $DS2$.

Lemma 6 *Let G be a type variable free query, let D be a program clause whose free variables are included in $x_1, \ldots, x_n$ and let $\mathcal{P}_1$ and $\mathcal{P}_2$ be programs that between them contain a formula of the form $D[c_1'/x_1, \ldots, c_n'/x_n]$ where, for $1 \leq i \leq n$, c_i' is a nonlogical constant of the same type as x_i. Further, for $1 \leq i \leq n$, let c_i be a nonlogical constant of the same type as c_i' that do not appear in D. Finally, let $\mathcal{P}_1'$, $\mathcal{P}_2'$ and G' be obtained from $\mathcal{P}_1$, $\mathcal{P}_2$ and G, respectively, by replacing, for $1 \leq i \leq n$, c_i with c_i'. Now, if $\mathcal{P}_1, D[c_1/x_1, \ldots, c_n/x_n], \mathcal{P}_2 \longrightarrow G$ has a derivation of length l in $DS2$, then there must also be a derivation in $DS2$ for $\mathcal{P}_1', \mathcal{P}_2' \longrightarrow G'$ that is of length l or less.*

Proof. We prove the lemma by an induction on the length of the derivation in DS2 of the first sequent. If this derivation is of length 1, it must have been obtained by using the SUCCESS rule. Now, if G is equal to an instance of $D[c_1/x_1, \ldots, c_n/x_n]$, then G' must be equal to an instance of $D[c_1'/x_1, \ldots, c_n'/x_n]$. Further, if G is an instance of a clause in $\mathcal{P}_1$ or in $\mathcal{P}_2$ it must be the case that G' is an instance of a clause in $\mathcal{P}_1'$ or in $\mathcal{P}_2'$. From these observations it follows that the SUCCESS rule is applicable to $\mathcal{P}_1', \mathcal{P}_2' \longrightarrow G'$ as well and so this sequent also must have a derivation of length 1.

Suppose now that the derivation of $\mathcal{P}_1, D[c_1/x_1, \ldots, c_n/x_n], \mathcal{P}_2 \longrightarrow G$ is of length $(l+1)$. We assume that the requirements of the lemma are satisfied by all sequents that have derivations of length l or less and show this must also be the case for the sequent being considered. The argument proceeds by examining the possible cases for the first rule used in the derivation in question.

Let us assume that this rule is an AND. In this case G must be of the form $G_1 \wedge G_2$ and there must be derivations of length l or less for the sequents $\mathcal{P}_1, D[c_1/x_1, \ldots, c_n/x_n], \mathcal{P}_2 \longrightarrow G_1$ and $\mathcal{P}_1, D[c_1/x_1, \ldots, c_n/x_n], \mathcal{P}_2 \longrightarrow G_2$. By hypothesis, there are derivations of length l or less for $\mathcal{P}_1', \mathcal{P}_2' \longrightarrow G_1$ and $\mathcal{P}_1', \mathcal{P}_2' \longrightarrow G_2$. Using these derivations together with an AND rule, we obtain one of length $l+1$ or less for $\mathcal{P}_1', \mathcal{P}_2' \longrightarrow G_1' \wedge G_2'$. Now, G' must be equal to the formula $G_1' \wedge G_2'$. Thus the desired conclusion is obtained in this case.

Arguments similar to that for AND can be supplied for the cases when OR or INSTANCE is the first rule used. In the case that GENERIC is used, G must be of the form $\forall y G_1$ and there must be a derivation of length l for

$$\mathcal{P}_1, D[c_1/x_1, \ldots, c_n/x_n], \mathcal{P}_2 \longrightarrow G_1[a/y]$$

for some nonlogical constant a of the same type as y that does not appear in G, $D[c_1/x_1, \ldots, c_n/x_n]$ or in the formulas in $\mathcal{P}_1$ and $\mathcal{P}_2$. We can almost use an argument similar to that employed for AND. The only problem is that a might be identical to some c_i' for $1 \leq i \leq n$. However, the following fact is easily seen: a derivation of length l for a sequent Ξ can be transformed into one of identical length for a sequent obtained from Ξ by replacing all occurrences of a nonlogical constant b with some other (nonlogical) constant of the same type. Using this together with the "newness" condition on a, we may assume that a is distinct from all the c_i's. The argument in this case can then be completed without trouble.

In the case that the first rule employed is BACKCHAIN, a combination of the observations used for SUCCESS and AND must be employed. In particular, let G_1' be the result of replacing, for

$1 \leq i \leq n$, all occurrences of c_i by c_i' in G_1. Now, if $G_1 \supset G$ is an instance of $D[c_1/x_1, \ldots, c_n/x_n]$, then $G_1' \supset G'$ must be an instance of $D[c_1'/x_1, \ldots, c_n'/x_n]$. Further, if $G_1 \supset G$ is an instance of a program clause in $\mathcal{P}$, then $G_1' \supset G'$ must also be an instance of the same clause. Finally, using the hypothesis, if $\mathcal{P}_1, D[c_1/x_1, \ldots, c_n/x_n], \mathcal{P}_2 \longrightarrow G_1$ has a derivation of length l, then $\mathcal{P}_1', \mathcal{P}_2' \longrightarrow G_1'$ has a derivation of length l or less. Using these various facts, it is easily seen that if the first rule used in the derivation for $\mathcal{P}_1, D[c_1/x_1, \ldots, c_n/x_n], \mathcal{P}_2 \longrightarrow G$ is BACKCHAIN, then a derivation can be provided for $\mathcal{P}_1', \mathcal{P}_2' \longrightarrow G'$ in which the last rule is once again a BACKCHAIN and, further, this derivation will satisfy the length requirements.

Suppose now that the first rule used is AUGMENT$'$ and that case (i) of this rule is the applicable one. Then G must be of the form $(\exists y_1 \ldots \exists y_m D_1) \supset G_1$ and further, no formula of the form $D_1[a_1'/y_1, \ldots, a_m'/y_m]$ must appear in $\mathcal{P}_1, D[c_1/x_1, \ldots, c_n/x_n], \mathcal{P}_2$. By assumption, there is a derivation of length l for

$$D_1[a_1/y_1, \ldots, a_m/y_m], \mathcal{P}_1, D[c_1/x_1, \ldots, c_n/x_n], \mathcal{P}_2 \longrightarrow G_1$$

where, for $1 \leq i \leq m$, a_i is a constant of appropriate type and meeting the needed requirements of newness. By an argument similar to that used in the case of BACKCHAIN, we can assume that the a_js are distinct from the c_is and the c_i's. Then, using the induction hypothesis, there must be a derivation of length l or less for

$$D_1'[a_1/y_1, \ldots, a_m/y_m], \mathcal{P}_1', \mathcal{P}_2' \longrightarrow G_1'$$

where D_1' and G_1' are obtained from D_1 and G_1 by the replacement, for $1 \leq i \leq n$, of c_i by c_i'. Now, if a formula of the form $D_1[a_1'/y_1, \ldots, a_m'/y_m]$ did not appear in $\mathcal{P}_1$ or $\mathcal{P}_2$, then one of the form $D_1'[a_1'/y_1, \ldots, a_m'/y_m]$ cannot appear in $\mathcal{P}_1'$ or $\mathcal{P}_2'$. Thus, the derivation of the indicated sequent can be used together with an AUGMENT$'$ rule to obtain one for $\mathcal{P}_1', \mathcal{P}_2' \longrightarrow (\exists y_1 \ldots \exists y_m D_1') \supset G_1'$; a newness condition has to be satisfied by $a_1, \ldots, a_m$ for the AUGMENT rule to be used, but this can be seen to be the case, using particularly the assumption of distinctness from the c_i's. The derivation of the last sequent is obviously of length $(l+1)$ or less. Observing that $(\exists y_1 \ldots \exists y_m D_1') \supset G_1'$ is the same formula as G', the lemma is seen to hold in this case.

The only situation remaining to be considered is that when the first rule corresponds to case (ii) of AUGMENT$'$. The argument in this case is similar to that employed for case (i) of the same rule. The details are left to the reader.

□

Using the above lemma we now show the equivalence of $DS1$ and $DS2$ from the perspective of derivability of sequents of the kind we are interested in.

Theorem 7 *Let $\mathcal{P}$ be a program and let G be a type variable free query. There is a derivation for $\mathcal{P} \longrightarrow G$ in $DS1$ if and only if there is a derivation for the same sequent in $DS2$.*

Proof. Consider first the forward direction of the theorem. The only reason why the derivation in $DS1$ might not already be one in $DS2$ is because the AUGMENT rule that is used is in some cases not an instance of the AUGMENT$'$ rule. Consider the last occurrence of such a rule in the derivation. In this case, a derivation is constructed for a sequent of the form $\mathcal{P}' \longrightarrow (\exists x_1 \ldots \exists x_n D') \supset G'$

from one for the sequent $D'[c_1/x_1, \ldots, c_n/x_n], \mathcal{P}' \longrightarrow G'$, where the c_is are appropriately chosen constants. Given that we are considering the last occurrence of an errant rule, the derivation for the latter sequent must be one in $DS2$ as well. Since the application of the AUGMENT rule being considered does not conform to the requirements of the AUGMENT′ rule, it must be the case that, for some choice of constants $c'_1, \ldots, c'_n$, $D'[c'_1/x_1, \ldots, c'_n/x_n]$ appears in $\mathcal{P}'$. But then, using Lemma 6 and the fact that the constants $c_1, \ldots, c_n$ must not appear in G' or in the formulas in $\mathcal{P}'$, we see that $\mathcal{P}' \longrightarrow G'$ has a derivation in $DS2$. Using this derivation together with case (i) of the AUGMENT′ rule, we obtain a derivation in $DS2$ for the original sequent, *i.e.*, for $\mathcal{P}' \longrightarrow (\exists x_1 \ldots \exists x_n D') \supset G'$. We repeat this form of argument to ultimately transform the derivation in $DS1$ for $\mathcal{P} \longrightarrow G$ into one in $DS2$.

To show the theorem in the reverse direction, we observe the following fact: for any program $\mathcal{P}'$, type variable free query G' and program clause D', if $\mathcal{P}' \longrightarrow G'$ has a derivation in $DS1$, then $D', \mathcal{P}' \longrightarrow G'$ also has a derivation in $DS1$. Now, a derivation in $DS2$ may not be a derivation in $DS1$ only because case (i) of AUGMENT′ was used in some places. However, this can be corrected by using the observation just made. In particular, we consider the last occurrence of an errant rule in the derivation and convert it into an occurrence of the AUGMENT rule by using the above fact. A repeated use of this transformation yields the theorem.

□

An easy consequence of the above theorem is the following:

Corollary 8 *Let $\mathcal{P}$ be a program and let G be a query. The set of answers to G in the context of $\mathcal{P}$ is independent of whether rules in $DS1$ or in $DS2$ are used in constructing derivations.*

We have thus shown that, from the perspective of solving queries and computing answers, it is immaterial whether the rules in $DS1$ or those in $DS2$ are used to construct derivations. By virtue of Proposition 4, we can in fact use the notion of intuitionistic derivability for the purpose of analyzing programs in our language while using the rules in $DS2$ to carry out computations. At a pragmatic level, there is a definite benefit to using the AUGMENT′ rule instead of the AUGMENT rule in solving queries, since considerable redundancy in search can be eliminated by this choice. We use this observation to yield a more viable implementation of module implication and of the *import* statement in the next section. We note that another approach to controlling the redundancy arising out of the module semantics is suggested in [12]. However, this approach is less general than the one considered here in that it applies only to *import* statements and not to module implications. Moreover, the correctness of the approach is only conjectured in [12]. The observations in this section can be used in a straightforward fashion to verify this conjecture.

7 An Improved Implementation of Modules

We now consider an implementation of our language that uses the AUGMENT′ rule instead of the AUGMENT rule whenever possible. Under the new rule, solving an goal of the form $(\exists x_1 \ldots \exists x_n D) \supset G$ requires checking if there is already a clause of the form $D[c_1/x_1, \ldots, c_n/x_n]$ in the program. Clearly an efficient procedure for performing this test is a key factor in using

the changed rule in an actual implementation. It is difficult to achieve this goal in general. One problematic case is when the goal $(\exists x_1 \ldots \exists x_n D) \supset G$ arises as part of a larger goal and D contains variables that are bound only in this larger context. Instantiations for these variables may be determined in the course of execution, thus making it difficult to perform the desired test by a simple runtime operation. In fact, the device of delaying instantiations might even make it impossible to determine the outcome of the test at the time the implication goal is to be solved because "clauses" in the program might contain logic variables. An example of this kind was seen in Section 3. The attempt to solve the goal $\exists L(rev\ 1 :: 2 :: nil\ L)$ resulted there in the clause $(rev_aux\ nil\ L)$ being added to the program. The precise shape of this clause clearly depends on the instantiation chosen for L. A test of the sort needed by AUGMENT$'$ cannot be performed with regard to this clause prior to this shape being determined.

The above discussion demonstrates that the optimization embodied in the AUGMENT$'$ rule can be feasibly implemented only relative to a restricted class of program clauses, namely, clauses that do not contain logic variables. Of particular interest from this perspective is a statically identifiable closed E-formula that has the potential for appearing repeatedly in the antecedent of implication goals. Given such a formula E, a mark can be associated with it that records whether or not the current goal is dynamically embedded within the invocation of an implication goal of the form E => G. If it is so embedded and if the current goal is itself of the form E => G', then, in accordance with the AUGMENT$'$ rule, the computation can proceed directly to solving G' without affecting additions to the program.

The dynamic semantics of module implication provides a particular case of the kind of formula discussed above, namely the (closed) E-formula identified with a module. Thus, assume that we are trying to solve the goal M ==> G. If we know that the module M has already appeared in the antecedent of a module implication goal within which the current one is dynamically embedded, then no enhancements to the program need be made. The implementation scheme presented in 5 provides a setting for incorporating this test in an efficient manner. The essential idea is that we include an extra field called *added* in the record in the global table corresponding to each module. This field determines whether or not the clauses in a particular module have been added to the program in the path leading up to the current point in computation. When the goal M ==> G is to be solved, the *added* field for M's entry in the global table is checked. If this indicates that the clauses in M has not previously been added, then the addition is performed and the status of the field is changed. Otherwise the computation proceeds directly to solving G.

While the idea described above is simple, some details have to be paid attention to in its actual implementation. One issue is the action to be taken on the completion of a module implication goal. At a conceptual level, the successful solution of the goal M ==> G must be accompanied by a removal of the code for M; this is accomplished in our earlier scheme by the instruction *pop_impl_point*. However, given the current approach, an actual removal must complement only an actual addition. To facilitate a determination of the right action to be taken, the *added* field is implemented as a counter rather than as a boolean. This field is initialized to 0. Each time a module is conceptually added to the program context, its *added* value is incremented. A conceptual removal similarly causes this value to be decremented. An actual removal is performed only when the counter value reaches 0.

The second issue that must be considered is the effect of backtracking. As we have noted, this operation might require a return to a different program context. An important characteristic of a program context now is the status of the *added* fields, and backtracking must set these back to values that existed at an earlier computation point. To permit an accomplishment of this resetting action, changes made to this field must be trailed. A naive implementation would trail the old value every time a change needs to be made, *i.e.*, every time a module is added or removed. A considerable improvement on this can be obtained by trailing a value only if there is a possibility to return to a state in which it is operative. Thus consider a goal of the form

m ==> $(G1, (m$ ==> $G2))$

When the *added* field for m is incremented for the second time, there is a need to trail the old value only if unexplored alternatives exist in the attempt to solve $G1$. There is a simple way to determine this within a WAM-like implementation. Let us suppose we record the address of the most recent choice point at the time of processing the outermost (module) implication in the global table entry corresponding to m. Now, when the embedded implication is processed, we compare the address of the current most recent choice point with the recorded value. There is a backtracking point in the solution of $G1$ only if the first is greater than the second. Similarly, consider the decrement that is made to the *added* field when a goal of the form m ==> G is completed. The old value needs to be trailed only if choice points exist within the solution for G. A test identical to that described above suffices to determine whether this is the case.

In order to implement the above idea, one more field must be added to the entries in the global table for modules, *i.e.*, one that records the most recent choice point prior to the latest change to the *added* field. This field is called *mrcp* and is initialized to the bottom of the stack. Notice that this field needs to be updated each time *added* has to be trailed, and this change must also be trailed. Accordingly, each cell in the trail introduced for managing the *added* values contains three items: the name of a module, the old value of *added*, and the old contents of the *mrcp* field. Pointers to this trail must be maintained in choice points and the trail must be unwound in the usual fashion upon backtracking. Module implication is compiled as before, although the interpretation of *push_impl_point* m and *pop_impl_point* m changes. In particular, these can be understood as though they are invocations to the procedures *pushimpl*(m) and *popimpl*(m) that are presented in pseudo-code fashion in Figure 4. In this code we write *m.mrcp* and *m.added* to denote, respectively, the *mrcp* and *added* fields in the global table entry corresponding to the module m. We also recall that the B register in the WAM setting indicates the most recent choice point.

There is an auxiliary benefit to two fields that has been added under the present scheme to the records in the global table. As mentioned in Section 5, our implementation permits modules to be loaded on demand, and hence does not require all modules to be available in main memory during a computation. A question that arises is whether modules can also be unloaded to reclaim code space. This unloading must be done carefully because a module not currently included in the program context might still be required because of the possibility of backtracking. A quick check of whether a module can be unloaded is obtained by examining the two new fields in the global table entry for a module. If the *mrcp* field points to the bottom of the stack and *added* is 0, then the module is not needed and can be unloaded.

```
pushimpl(m)
begin
    if m.added = 0
    then create an implication point record for m;
    if m.mrcp < B
    then
    begin
        trail ⟨m, m.mrcp, m.added⟩;
        m.mrcp := B;
    end;
    m.added := m.added + 1
end;

popimpl(m)
begin
    if m.mrcp < B
    then
    begin
        trail ⟨m, m.mrcp, m.added⟩;
        m.mrcp := B;
    end;
    m.added := m.added − 1;
    if m.added = 0 then
        Set I to most recent implication point
        in record pointed to by I
end
```

Figure 4: Adding and Removing Modules from Program Contexts

The implementation of the dynamic effects of *import* can, in principle, be left unchanged. However, a significant efficiency improvement can be obtained by noting the following: once a clause from a module m has been used by virtue of the BACKCHAIN rule, there is no further need to check if the modules imported by m have been added to the program context. To utilize this idea, we include two more fields in each implication point record:

(i) A field called *backchained* that records the number of times a clause from the module to which the implication point record corresponds has been backchained upon.

(ii) A field called *mrcp* that records the most recent choice point prior to the last change to *backchained.*

When the implication point record is created, the *backchained* field is initialized to 0 and the *mrcp* field is set to point to the bottom of the stack. Whenever a clause from a module corresponding to the implication point record is backchained upon, a conceptual addition of the imported modules must be performed. An actual addition must be contemplated within the present scheme only if the *backchained* field is 0. In any case, this field is incremented before the "body" of the clause is invoked. The increment to *backchained* is complemented by a decrement when the clause body has been successfully solved. Finally, an actual removal of the imported modules from the program context must be contemplated only when *backchained* becomes 0 again. For the purpose of backtracking, it may be necessary to trail an old value of *backchained* each time this field is updated. The *mrcp* field is useful for this purpose. Essentially, we compare this field with the address of the current most recent choice point, obtained in the WAM context from the `B` register. If the latter is greater than the *mrcp* field, then the old value of *backchained* must be trailed. This action must also be accompanied by a trailing of the existing *mrcp* value and the update of this field to the address of the current most recent choice point.

The rationale for the various actions described for handling imports is analogous to that in the case of module implication, and should be clear from the preceding discussions. At a level of detail, another trail is needed for maintaining the old values of the *backchained* and *mrcp* fields. The cells in this trail correspond once again to triples: the address of the relevant implication point record and the *backchained* and *mrcp* values. Pointers to this trail must also be maintained in choice points and backtracking must cause the trail to be unwound. The compilation of clauses in modules is performed as before: the code produced for the body of a clause in module m must be embedded within the instructions *push_import_point* m and *pop_import_point* m. These instructions can be understood as though they are invocations to the procedures *pushimport*(m) and *popimport*(m) that are presented, in pseudo-code fashion, in Figure 5. Use is made in these procedures of a register called `CI` that points within our implementation to the implication point record from which the clause currently being considered is obtained. Further, we write `CI`.*mrcp* and `CI`.backchained to denote the *mrcp* and *backchained* fields in the implication point record that *CI* points to.

It is important to note that once a clause from a module has been backchained upon, the two instructions *push_import_point* and *pop_import_point* incur very little overhead with respect to clauses in that module. In particular, at most two tests, a trailing and two updates are necessary

```
pushimport(m)
begin
    if CI.backchained = 0
    then call load_imports for m
    if CI.mrcp < B
    then
    begin
        trail ⟨CI,CI.mrcp,CI.backchained⟩;
        CI.mrcp := B;
    end;
    CI.backchained := CI.backchained + 1
end;

popimport(m)
begin
    if CI.mrcp < B
    then
    begin
        trail ⟨CI,CI.mrcp,CI.backchained⟩;
        CI.mrcp := B;
    end;
    CI.backchained := CI.backchained − 1;
    if CI.backchained = 0 then
        invoke unload_imports for m
end
```

Figure 5: Adding Imported Modules to a Program Context

for each instruction. This is much less work than the creation of implication point records that was necessary under a direct implementation of the operational semantics. Further, this overhead appears to be acceptable even if these instructions are executed repeatedly.

We consider an example to illustrate the manner in which redundancy is controlled within the changed implementation. Let us assume that the modules $m0$, $m1$ and $m2$ are defined as below.

module $m0$.	*module* $m1$.	*module* $m2$.
import $m1, m2$.	*import* $m2$.	*kind* i *type*.
type $p\ i \rightarrow o$.	*type* $q\ i \rightarrow o$.	*type* $a\ i$.
$(p\ X)$:- $(q\ X), (t\ X)$.	$(q\ X)$:- $(r\ X)$.	*type* $b\ i$.
$(r\ X)$:- $(s\ X)$.		$(r\ a)$.
		$(s\ b)$.
		$(t\ b)$.

The attempt to solve the goal $m0$ ==> $(p\ X)$ is presented below. We augment the linear format of Section 4 as follows in this presentation: Each module in the program context is presented by a pair consisting of its name and the value of the *backchained* field in the implication point record created for it. At the end of each line, a list of pairs is presented that indicates module names and the values of the *added* field in the global table entry for each of them.

$(m0,0)$	?-	$(p\ X)$			$[(m0,1),(m1,0),(m2,0)]$
$(m2,0),(m1,0),(m0,1)$	?-	$(q\ X)$			$[(m0,1),(m1,1),(m2,1)]$
$(m2,0),(m1,1),(m0,1)$	?-	$(r\ X)$	X <- a	$SUCC$	$[(m0,1),(m1,1),(m2,2)]$
$(m2,0),(m1,0),(m0,1)$	?-	$(t\ a)$	$FAIL$		$[(m0,1),(m1,1),(m2,1)]$
$(m2,0),(m1,1),(m0,1)$	?-	$(r\ X)$			$[(m0,1),(m1,1),(m2,2)]$
$(m2,0),(m1,1),(m0,2)$	?-	$(s\ X)$	X <- b	$SUCC$	$[(m0,1),(m1,1),(m2,2)]$
$(m2,0),(m1,0),(m0,1)$	?-	$(t\ b)$	$SUCC$		$[(m0,1),(m1,1),(m2,1)]$

An interesting point to note in this computation is that the clause $(r\ a)$ in module $m2$ is used only once in solving the subgoal $(r\ X)$ even though there are conceptually two copies of $m2$ in the program context when the subgoal is invoked. Similarly, an attempt to find another solution to the query will fail, even though the same solution can be found five more times under a naive interpretation of the given semantics.

8 Conclusion

We have examined a notion of modules for the logic programming language λProlog in this paper. The notion considered provides a means for structuring the two components that determine programs in this language: the type, kind and operator declarations and the procedure definitions. Using a module typically involves making its contents available in some other context. As explained in some detail, this operation has static and dynamic effects within λProlog. Our focus here has been on the implementation of the dynamic aspects of modules. At a level of detail, we have

proposed an implementation method that is based on a WAM-like machine and that has several interesting features:

(i) It supports the idea of compiling modules separately. In particular, the compilation of a module produces WAM-like code based on only the program clauses appearing in the module.

(ii) Interpreting a logical operation as a primitive for linking a module into a given program context, it uses a compilation process to generate linking code and includes run-time structures for accomplishing the linking function.

(iii) Based on a theoretical analysis of this notion, it includes mechanisms for reducing redundancy inherent in the given dynamic semantics of the module feature. The redundancy check is based on a two-level test that in the usual situation can be carried out with very little overhead.

There are several significant enrichments to a Prolog-like language that are embodied in λProlog in addition to the module feature. A complete implementation of this language must include mechanisms for dealing with all these features. As mentioned already, a detailed consideration has been given to the features other than the module notion elsewhere, resulting in an abstract machine for the core language described in Section 3. An actual implementation of this machine is currently being undertaken. The mentioned machine is entirely compatible with the ideas for handling modules that are presented in this paper and we plan to include these ideas within our implementation effort in the near future.

References

[1] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[2] Conal Elliott and Frank Pfenning. eLP, a Common Lisp Implementation of λProlog. Implemented as part of the CMU ERGO project, May 1989.

[3] Conal Elliott and Frank Pfenning. A semi-functional implementation of a higher-order logic programming language. In Peter Lee, editor, *Topics in Advanced Language Implementation*, pages 289–325. MIT Press, 1991.

[4] Elsa L. Gunter. Extensions to logic programming motivated by the construction of a generic theorem prover. In Peter Schroeder-Heister, editor, *Extensions of Logic Programming: International Workshop, Tübingen FRG, December 1989*, pages 223–244. Springer-Verlag, 1991. Volume 475 of *Lecture Notes in Artificial Intelligence*.

[5] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinatory Logic and Lambda Calculus.* Cambridge University Press, 1986.

[6] Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[7] Keehang Kwon, Gopalan Nadathur, and Debra Sue Wilson. Implementing polymorphic typing in a logic programming language. Submitted, August 1992.

[8] Evelina Lamma, Paola Mello, and Antonio Natali. The design of an abstract machine for efficient implementation of contexts in logic programming. In *Sixth International Logic Programming Conference*, pages 303–317, Lisbon, Portugal, June 1989. MIT Press.

[9] Dale Miller and Gopalan Nadathur. λProlog version 2.7. Distributed in C-Prolog and Quintus Prolog source code, August 1988.

[10] Dale Miller. Lexical scoping as universal quantification. In *Sixth International Logic Programming Conference*, pages 268–283, Lisbon, Portugal, June 1989. MIT Press.

[11] Dale Miller. A logical analysis of modules in logic programming. *Journal of Logic Programming*, 6:79 - 108, 1989.

[12] Dale Miller. A proposal for modules in λProlog. In *Workshop on the λProlog Programming Language*, Philadelphia, July 1992.

[13] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[14] Luís Monteiro and António Porto. Contextual logic programming. In *Sixth International Logic Programming Conference*, pages 284–299, Lisbon, Portugal, June 1989. MIT Press.

[15] Gopalan Nadathur and Frank Pfenning. The type system of a higher-order logic programming language. In Frank Pfenning, editor, *Types in Logic Programming*, pages 245–283. MIT Press, 1992.

[16] Gopalan Nadathur, Bharat Jayaraman, and Keehang Kwon. Scoping constructs in logic programming: Implementation problems and their solution. Submitted, May 1992.

[17] Gopalan Nadathur, Bharat Jayaraman, and Debra Sue Wilson. Implementation considerations for higher-order features. Submitted, November 1992.

[18] Gopalan Nadathur. A proof procedure for the logic of hereditary Harrop formulas. To appear in the *Journal of Automated Reasoning*.

[19] Richard O'Keefe. Towards an algebra for constructing logic programs. In *1985 Symposium on Logic Programming*, pages 152–160, Boston, 1985.

[20] D.T. Sannella and L.A. Wallen. A calculus for the construction of modular Prolog programs. *Journal of Logic Programming*, 12:147 - 178, January 1992.

[21] D.H.D. Warren. An abstract Prolog instruction set. Technical report, SRI International, October 1983. Technical Note 309.

λPROLOG IMPLEMENTATION OF RIPPLE-REWRITING: ABSTRACT

Chuck Liang
Computer Science Department
University of Pennsylvania
Philadelphia, PA 19104-6389 USA
liang@saul.cis.upenn.edu

Introduction

The wave-rippling theorem proving method was introduced by Alan Bundy et al, to guide the proofs of inductive theorems [1]. In implementing fragments of this work in λProlog[1], we hope to achieve the following:

1. To demonstrate that λProlog's higher ordered logic of hereditary Harrop formulas can provide a clear and declarative implementation of the wave-rippling method.
2. A better understanding of the wave-rippling method through this implementation.
3. To demonstrate the use of λProlog in specifying a meta language of tactics and tacticals used in controlling theorem proving.

There are two dimensions to consider in building this theorem prover: how much guidance (information concerning the manner and order of rule application) is inherent in the rewrite system itself, and how much control should be offered by the meta-level theorem proving mechanism. One must take care that control mechanisms of the theorem prover do not undermine the inherent automation of the rules themselves, and yet still provide the means to control different degrees of automatic rewrite.

Augmenting a rewrite system

Alan Bundy's wave rippling method is an attempt at annotating rewrite rules with *suggestions* on how a proof should be carried out [1]. This work was originally intended to solve inductive problems but can be used to proof other kinds of theorems as well. For example, the following rewrite rule for the successor operator in arithmetic: $s(x) + y = s(x + y)$ could be annotated with *wave fronts* and become $(wave\ s\ x) + y = (wave\ s\ (x + y))$. Here, s is a constructor called the *wave front* and x and $(x + y)$ are contents of the *wave holes.* The aim is to "ripple" the wave fronts outward. If we were to prove by induction on x that $(x + y) + z = x + (y + z)$, then the inductive

[1]Three problem domains: arithmetics, lists and sum-series are implemented in our system. They are representitive of the spectrum of rippling applications. More domains will be addressed later.

conclusion is $(s(x)+y)+z = s(x)+(y+z)$. We would like to show that this inductive conclusion follows from the inductive hypothesis. This task is made easier if the conclusion was annotated with wave fronts: $((wave\ s\ x)+y)+z = (wave\ s\ x)+(y+z)$. After three application of the wave ripple rule $(wave\ s\ x)+y \longrightarrow (wave\ s\ (x+y))$ (twice on the left, once on the right), we obtain $(wave\ s\ (x+y)+z)) = (wave\ s\ x+(y+z))$. Now the wave fronts have been *fully rippled,* and what's inside the wave fronts is exactly the inductive hypothesis. Another ripple rule, using the fact that s is injective, eliminates the outer s wave-fronts, and the proof is complete. This illustrates how wave front annotations can guide the construction of a proof. Were there no wave fronts, the rewrite from the inductive conclusion to the hypothesis could take any of a number of possible routes, i.e., the search space would be too huge to expect efficient proofs to result.[2]

Representing Rippling in λProlog

A wave front is represented as a higher-order lambda term. The bound variable represents the positions of the wave hole. If t1 is the type of expression in question (integer in the above example), then the wave front will have type t1 → t1. The content of the wave holes is another t1 expression. The wave expression constructor "wave" have type (t1 → t1) → t1 → t1. The entire expression (*wave Front Hole*) is again of type t1[3]. This representation means that the wave hole could have several occurrences inside the front since the bound variable could have several occurrences, but the content of the hole has to be the same for each occurrence. We chose not to have wave-fronts of form $(wave\ (\lambda x.\lambda y.(P\ x\ y))\ H1\ H2)$ because of typing problems. Such composite wave forms can be broken up into separate instances: $(wave\ (\lambda x.(P\ x\ H2))\ H1)$ and $(wave\ (\lambda y.(P\ H1\ y))\ H2)$, each with its own set of rippling rules. This representation is arguably more desirable because we now have more control over which part of the wave to ripple.

The choice of using lambda terms to represent wave fronts is a natural one. A first order representation will have to contend with locating the wave holes inside the wave fronts, and with the well-formedness of expressions. Lambda abstraction makes these issues trivial. An expression annotated with waves should be recoverable, i.e, we need to be able to know what is the *real* expression being considered (wave-fronts, after all, adds no more expressive power to a rewrite system). A first order representation will require an explicit de-annotation procedure to surgically remove wave fronts. The higher order representation has implemented the constructor *wave* as a kind of *delayed* function application. Therefore, de-annotation of a wave expression $(wave\ F\ H)$ is easily accomplished with $(F\ H)$. Sometimes it is also desirable to merge two wave fronts :

$$(wave\ F\ (wave\ G\ H))$$

into a single wave front: the composition of F and G. But function composition is expressed naturally in our system of lambda terms. The merged wave expression is $(wave\ (\lambda x.(F\ (G\ x)))\ H)$. No such obvious method exists in first order systems that would allow this kind of composition.

[2]The reader may wish to see [1] for a more complete background on rippling.

[3]A generic "wave" is used here for simplicity. In the actualy implementation there are different wave constructors for each type, i.e, wavei for integers, wavel for lists. λProlog does not support dependent types (as opposed to Elf), which would allow a polymorphic definition of wave that still ensures that the type of a wave is the same as the type of terms it annotates.

The problem of wave annotation also illustrates the natural choice of using a higher-order representation. In proving a problem of form $(\forall A)$ (such as $(\forall\lambda x.(x = x + 0))$), we need to specify the inductive basis, the inductive hypothesis and the inductive conclusion, which is annotated with wave fronts. This can be done easily in our representation. Let I be an inductive constructor, for example the successor function in the integer case), then the base case goal is simply $(A\ 0)$, the inductive hypothesis will be $(A\ n)$ for some arbitrary n, and the inductive conclusion will be $(A\ (wave\ s\ n))$ or equivalently $(A\ (wave\ (\lambda x.(x + 1))\ n))$.[4] The lambda term representation of wave fronts allows the use of function application to implement substitution, which is a tedious and (because of the danger of bound variable capture) potentially unsafe task in first order systems.

Implementing the theorem prover

Idealy, a rewrite system annotated with wave fronts should need no further support to construct correct and efficient proofs. We should need only initiate the rippling process. However, it will be naive to assume that rippling alone can produce efficient proofs. There are several different types of rippling rules and the order they are applied is important. Sometimes it is also preferable to perform normal rewriting, such as normalization, rather than applying rippling rules. It is therefore still necessary to support the wave-augmented rewrite system with an underlining theorem prover.

It has been argued that effective theorem provers can not be specified in Prolog because of the limitations of first-order Horn clause logic, and because the naive, depth-first backtracking method of Prolog interpreters prohibits more elaborate proof-search methods. However, Amy Felty, in [4,5] have demonstrated that this criticism of Prolog is invalid. Prolog's internal mechanism may be naive, but Prolog can be used to define a meta-level language of tactics, which can provide control over the theorem proving process independently of Prolog's internal search mechanisms. Prolog is used as the meta-language of the meta-language. Much of limitations of first-order Horn clause prolog can also be solved by the more expressive, higher-ordered hereditarily Harrop formulas of λProlog.

Our purpose is to implement a rich tactic system that would give the user the choice of varying degrees of control over the theorem proving process. The system can be specified to attempt to prove something automatically, or be used as an interactive proof-editor.

Theorem proving rules and methods are implemented by declaring "tactics". Tactics can be combined using a language of "tacticals." The following set of tacticals are defined following Felty [4].

```
apply_tac idtac A A.
apply_tac (then T1 T2) A C :- apply_tac T1 A B, apply_tac T2 B C.
apply_tac (orelse T1 T2) A C :- apply_tac T1 A C; apply_tac T2 A C.
apply_tac (try T) A B :- apply_tac (orelse T idtac) A B.
apply_tac (repeat T) A B :-
        apply_tac (orelse (then T (repeat T)) idtac) A B.
```

[4]In the inductive proof case the initial wave front is always the inductive operator (successor for integers, cons for lists), but rippling can also be used for non-inductive proofs (see [3]), in which case the initial annotation of wave fronts is much more difficult, and requires careful higher-order manipulations.

The purpose of a tactic is to advance the theorem proving process by one or more steps. **apply_tac** is given a tactic name and the current goal or form of the problem, and gives the updated goal, or the result of applying the tactic on the current goal. The aim is to reduce the initial goal (the theorem to be proved) to `truegoal`, which represents triviality. `idtac` is the most simple tactic in leaving the problem unchanged. The `try` tactic prevents failure by returning the same goal should the tactic fail. `repeat` repeatedly applies a tactic until it fails. `then` and `orelse` are self-explanatory.

These tacticals form the core of the meta-language of tactics. They are used to define other, more complicated tactics and tacticals. They have a natural declaration in prolog (in fact first-order prolog), and yet greatly extends the ability of prolog by providing more flexible control over goal search. For example, `(repeat (orelse (tactic1 (Then tactic2 tactic3))))` can be used to repeatedly transform a goal using either tactic1 or sequences of tactic2 and tactic3.

Rewrite rules are implemented as tactics. Each tactic can be viewed as the implementation of one or more rewrite rules. They are organized as follows:

Primitive normalization rules. These include rules such as $(x + 0 = x)$ for arithmetics and basic list equalities such as $(append\ a\ nil) = a$. Tactics are defined to implement these rules. Each tactic applies a primitive rule exactly once. We explicitly prevent exhaustive application to provide the option of precise control of rewriting through the tactics. Tactics can be exhaustively applied using the `repeat` tactical.

Special normalization rules. Additional constructors, such as user defined functions, need their own set of rules and corresponding tactics. For example, the function `reverse` for reversing lists will have a set of rewrite rules representing the functional evaluation rules for `reverse`. They are kept separate from the other normalization rules; again, to provide precise control over rewriting. The indiscriminate application of both primitive and special rules is achieved using the `orelse` tactical.

Wave rippling rules. These rules/tactics are annotated with a direction: outward (the standard type), sideways or inward. This is the core of the theorem prover.

"Proof Plans." A proof-plan is a clause that implements a series of procedures for carrying out proofs for a certain type of theorem. These procedures include tactics, but also other facilities. We could implement proof plans as composite tactics but choose not to, because, theoretically, there could be a meta-language of proof-planning separate from the meta-language of tactics. For example, integer induction can be specified as the following proof plan:

Prove base case using normalization,
Annotate inductive conclusion with wave-fronts,
Exhaustively apply ripple outward,
Apply normalization to the result,
Match result with inductive hypothesis.

Auxiliary tactics. These include, for example, equality, which makes use of normalization.

Issues in implementing ripple-rewrite in λProlog

The use of Higher-order unification must be precisely controlled to be effective. In this system, unification is only performed with variables on one side of the equation.[5] This reduces the otherwise unmanageable number of unifiers returned by the unification algorithm. The use of higher-order unification is limited to what was described earlier in systems such as arithmetics, which is not inherently higher-ordered. However, in inherently higher-ordered problem domains such as solving sum-series, which includes a notion of bound variables, higher-order unification becomes a necessity. For example, it is used in determining if a sum-series expression is independent of the index variable of the series. The implementation demonstrates the safe and effective use of higher-ordered terms and unification throughout.

λProlog's more expressive abilities allow the system to be defined without any use of extra-logical constructs found in first-order Horn clause Prolog systems such as cut, not, assert or call. Assert is replaced by the more logical $\Rightarrow$. For example, say we wanted to define a predicate to test if a formula is atomic. We could write:

```
atomic (and A B) :- !, fail.
atomic (or A B) :- !, fail.
...
atomic Anything.
```

Or we can explicitly write `(assert (atomic x))` for each new `x` we wish to be considered atomic. In λProlog, if in solving a goal G we wish to regard some `x` (usually introduced by the negative universal quantifier `pi`) as atomic, we simply write `(atomic x => G)`. Universally quantified formulas (at the object level) often require $\Rightarrow$ to place conditions on their bounded variables. This method is used in implementing the sum-series problem to test if an expression is free of sums (in which case the proof is complete).

The tactic system defined in λProlog further ensures that the system is purely logical. As an example, tactics do not recursively descend into a structure and perform rewrite on a subterm unless it is specifically predicated by the `descend` tactical. Non tactic-based systems often use cut (!) to explicitly control recursive descent. The `descend` tactical eliminates this reliance on extra-logical constructs. The use of `descend` and other search control tacticals also further illustrates the power of our tactic system in offering varying degrees of control over rewriting. For example, if the tactic `associativity` rewrites terms of the form $a + (b + c)$ into $(a + b) + c$, then the tactic `(descend associativity)` will apply associativity to a subterm if it fails at the outermost level; `(repeat (descend associativity))` will exhaustively rewrite an expression to eliminate at all levels terms of the form $a + (b + c)$. [6]

[5] In fact, usually with only one occurrence of an unbound logic variable. When expedient, β_0 redex of $L\lambda$ are used to further ensure smooth unification.

[6] Although not all implemented, other tacticals similar to descend that provide precise control over search and rule application can be easily defined given the core tactics. For example, although ideally, wave-front expressions should only be rewritten by rippling rules, rewrite rules can be made to descend into a wave-front with the `trans-wave` tactical.

Other Considerations

Although through this implementation we have formalized wave-rippling as a special form of rewrite, we have not shown how rippling rules should be selected. Bundy et al. have shown in [2] that deriving rippling rules directly from the recursive definition of functions in the style of the Boyer-Moore Theorem Prover is often not complete enough to guarantee the successful proof of a theorem. In general, a wave rule can be derived from any valid rewrite rule. Each regular rewrite rule can have a number of different wave-annotations, giving it several rippling interpretations. If all possible annotations are given, then this defeats the purpose of having ripple rules *guide* induction by limiting the number of choices in each rewrite step. The selection of rippling rules is clearly dependent on the problem domain. However, it may be possible to develop some kinds of standards of specifying ripple rules. For example, it is reasonable to hypothesize that only outward ripple rules are necessary in solving integer induction problems. We hope to study this problem further.

During the course of this implementation, many unclear issues in Bundy's presentation of rippling, such as the meaning and use of logic variables, are clarified through the declarative specification. We wish also to better understand how exactly existential quantifiers (object level) are treated.

The problem of typing needs to be addressed further. There are two typing issues to consider. First, do we put types at the meta level (using the typing system of λProlog) or do we define types at the object level (so that each rule and/or expression must be annotated with a type). The current system implements the first approach. Secondly, we wish our system to be polymorphic at least to some extent. For example, we wish to define lists of any type, not just, say, lists of integers. The first option is to use λProlog's own polymorphic typing system. But this will lead to problems in unification. The other option is to put the polymorphism into the tactic structure. Different rules of the same tactic are defined to permit the application of that tactic to different types. This is what has been adapted in the current system. For example, there are several rules for the equality tactic, each for a different type of equality. Neither of these issues has been completely resolved; they require further study.

Finally, a major goal of ours is to prove that our implementation is sound and at least to some extent, complete. Alan Bundy have already proved that rippling terminates if the ripple rules are used correctly.[7] Thus, we only need to show that our *implementation* terminates. We also need to show that if a rippling rule succeeds then the unannotated version of the rule is valid. Again, it is hoped that our higher-ordered, tactic-directed implementation will facilitate in such proofs. As mentioned earlier, the ease of de-annotating wave fronts from an expression to recover the real formula will be an important tool in our proofs. The simplicity, flexibility and expressiveness of our tactic system should aid in the proof of some kind of completeness.

[7] For example, if we know that only outward rippling is used, termination is relatively trivial. The other forms of rippling obviously complicates the problem greatly.

References

[1] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland and A. Smaill. Rippling: A Heuristic for Guiding Inductive Proofs. University of Edinburgh DAI Research Paper No. 567. February 1992.

[2] A. Bundy, F van Harmelen, J Hesketh, A. Smaill and A. Stevens. A Rational Reconstruction and Extension of Recursion Analysis. *The 11th International Joint Conference on Artificial Intelligence,* pages 359-365. Morgan Kaufmann 1989.

[3] A. Bundy, T. Walsh and A. Nunes. The Use of Proof Plans to Sum Series. University of Edinburgh DAI Research Paper. February 1992.

[4] A. Felty. Specifying And Implementing Theorem Provers in a Higher-Order Logic Programming Language. Ph.D. Thesis. University of Pennsylvania, August 1989.

[5] A. Felty. Implementing Tactics and Tacticals in a Higher-Order Logic Programming Language. August 1990. Submitted to Journal of Automated Reasoning.

Searching for Inductive Proofs in Second-Order Intuitionistic Logic

(Extended Abstract)

L. Thorne McCarty
Computer Science Department
Rutgers University
New Brunswick, NJ 08903, USA
mccarty@cs.rutgers.edu

1 Introduction

Several researchers have studied the problem of inductive reasoning about PROLOG programs, beginning with an early paper by Clark and Tarnlund [2]. The pioneering work of Kanamori and Seki [11] proposed an extended model of PROLOG execution and showed how this extended model could be used for program verification. A companion paper by Kanamori and Fujita [10] analyzed several techniques for the formulation of induction schemata and showed how two or more such schemata could be merged into one. These ideas have been extended and refined in a series of papers by Fribourg [4, 5, 6]. Other contributions include the work of Hsiang and Srivas [9] and Elkan and McAllester [3]

The biggest problem in all of this work seems to be: How to conjecture an appropriate induction schema? In this extended abstract, we will show how to formulate induction schemata in *second-order intuitionistic logic* [27], and how to search for these schemata in a logic programming language based on *embedded implications* [17, 18]. This is a report on work in progress, and it relies heavily on two concrete examples. One example ("Red and Green Blocks") is a variant of a familiar problem in common sense reasoning; the other example ("Naive Reverse") is a standard problem from the logic programming literature. We use these examples to illustrate our proposed technique, and to suggest that the ideas presented here are worth pursuing further. We will tackle the problem of inductive proofs in greater generality in a future paper.

Section 2 is a brief discussion of the theoretical foundations of our work, abstracted from [20]. The two examples are presented in Sections 3 and 4. Section 5 then outlines our current and future investigations into inductive reasoning.

2 Theoretical Background

The framework for our work is the language of *intuitionistic embedded implications* presented in [17, 18]. A similar language is studied in [23] and forms the basis of the λPROLOG program-

ming language [25]. Prior work on essentially the same language appears in [8, 7]. The class of intuitionistic embedded implications is given by:

Definition 2.1:

- An atomic formula is an embedded implication.
- If A is an atomic formula and $\mathcal{A}_1, \ldots, \mathcal{A}_k$ are embedded implications, then $A \Leftarrow \mathcal{A}_1 \wedge \ldots \wedge \mathcal{A}_k$ is an embedded implication.
- If $\mathcal{A}(x)$ is an embedded implication, then $(\forall x)\mathcal{A}(x)$ is an embedded implication.

This definition allows implications to be embedded to an arbitrary depth. However, we can restrict this definition to the class of *simple embedded implications* — in which implications are nested at most one deep — without any loss of expressive power, since arbitrary embeddings can be simulated by defining new atomic predicates using simple embedded implications exclusively.

It is easy to see that Definition 2.1 gives us a language which is equivalent, classically, to full first-order logic. However, interpreted intuitionistically, this language is a proper subset of first-order logic with interesting semantic properties [17]. Most significantly, a set of intuitionistic embedded implications $\mathcal{R}$ has the *disjunctive property* and the *existential property*. A disjunction of formulae, $\mathcal{A} \vee \mathcal{B}$, is entailed by $\mathcal{R}$ if and only if $\mathcal{R} \models \mathcal{A}$ or $\mathcal{R} \models \mathcal{B}$, and an existentially quantified formula, $(\exists \mathbf{x})\mathcal{A}(\mathbf{x})$, is entailed by $\mathcal{R}$ if and only if $\mathcal{R} \models \mathcal{A}(\mathbf{x})\theta$ for some ground substitution θ. Closely related is a proof-theoretic property, the existence of *linear proofs* [18] in which subgoals return definite answer substitutions to parent goals. (These are referred to as *uniform proofs* in [24].) Because of these properties, intuitionistic embedded implications provide a natural generalization of the class of *definite Horn clauses*.

But what if we wanted to represent *indefinite* information as well? A recent paper [21] suggests a novel approach to this question. Imagine a two-person communication situation in which the "speaker" applies a set of definite rules to a world of definite facts, and then reports some of these definite conclusions. Assume it is our job (as the "hearer") to make inferences about the actual state of the world, even though we have not observed it directly. McCarty and van der Meyden suggest that the correct way to formalize this problem is to *circumscribe* [15, 16] the defined predicates in the set of definite rules, and then to ask whether a certain *implicational goal* is entailed by the circumscription. In [21], the set of definite rules consists of a set of Horn clauses, but in [22] this model is extended to include actions defined by Horn clauses over a linear temporal order, and in [19] it is extended to include intuitionistic embedded implications as well. In all cases, the basic idea is to do *indefinite reasoning with definite rules*.

We now outline the machinery needed for this type of reasoning. Since we are working with intuitionistic logic, we need to use an intuitionistic version of the circumscription axiom. As in [21], we restrict our attention here to the circumscription of Horn clauses. Let $\mathcal{R}$ be a finite set of definite Horn clauses, and let $\boldsymbol{P} = \langle P_1, P_2, \ldots, P_k \rangle$ be a tuple consisting of the "defined predicates" that appear on the left-hand sides of the sentences in $\mathcal{R}$. Let $\mathcal{R}(\boldsymbol{P})$ denote the conjunction of the sentences in $\mathcal{R}$, with the predicate symbols in $\boldsymbol{P}$ treated as free parameters, and let $\mathcal{R}(\boldsymbol{X})$ be the

same as $\mathcal{R}(\boldsymbol{P})$ but with the predicate constants $< P_1, P_2, \ldots, P_k >$ replaced by predicate variables $< X_1, X_2, \ldots, X_k >$.

Definition 2.2: The *circumscription axiom* is the following sentence in second-order intuitionistic logic [27]:

$$\mathcal{R}(\boldsymbol{P}) \wedge (\forall \boldsymbol{X})[\mathcal{R}(\boldsymbol{X}) \wedge \bigwedge_{i=1}^{k} (\forall \mathbf{x})[X_i(\mathbf{x}) \Rightarrow P_i(\mathbf{x})] \Rightarrow \bigwedge_{i=1}^{k} (\forall \mathbf{x})[P_i(\mathbf{x}) \Rightarrow X_i(\mathbf{x})]]$$

We denote this expression by $Circ(\mathcal{R}(\boldsymbol{P});\boldsymbol{P})$, and we refer to it as "the circumscription of $\boldsymbol{P}$ in $\mathcal{R}(\boldsymbol{P})$." The circumscription axiom has the same intuitive meaning here that it has in classical logic. It states that the extensions of the predicates in $\boldsymbol{P}$ are as small as possible, given the constraint that $\mathcal{R}(\boldsymbol{P})$ must be true. Since the logic is intuitionistic, however, the axiom minimizes extensions at *every* state of *every* Kripke structure that satisfies $\mathcal{R}$.

Now let ψ be a Horn clause and let $\mathcal{Q}$ be a set of embedded implications. We are interested in the following *circumscriptive query problem*:

$$\mathcal{Q} \cup Circ(\mathcal{R}(\boldsymbol{P});\boldsymbol{P}) \models \psi?$$

We will discuss concrete instances of this problem in Sections 3 and 4. Since $Circ(\mathcal{R}(\boldsymbol{P});\boldsymbol{P})$ is a second-order sentence, however, one might ask: Is it possible to solve the circumscriptive query problem at all? The answer is: Yes, in certain special cases. Our analysis makes use of the concept of a *final Kripke model*, which is not discussed in this extended abstract. For more details, see [21, 19, 20].

First, if $\mathcal{R}$ is a set of *nonrecursive* Horn clauses, the solution is the same in intuitionistic logic as it is in classical logic [26, 13]. Let $Comp(\mathcal{R})$ denote Clark's Predicate Completion [1]. We then have the following result:

Theorem 2.3: Let $\mathcal{R}$ be a set of nonrecursive Horn clauses. Then $Circ(\mathcal{R}(\boldsymbol{P});\boldsymbol{P})$ is equivalent to $Comp(\mathcal{R})$.

For *recursive* Horn clauses, we initially restrict our analysis to a special case:

Definition 2.4: $\mathcal{R}$ is a *linear recursive definition* of the predicate A if it consists of:

1. A Horn clause with '$A(\mathbf{x})$' on the left-hand side and a conjunction of nonrecursive predicates on the right-hand side, and
2. A Horn clause that is linear recursive in A.

Let '$A(\mathbf{x}) \Rightarrow A^0(\mathbf{x})$' be the rule obtained from (1) by applying Clark's Predicate Completion. We say that '$A(\mathbf{x}) \Rightarrow A^0(\mathbf{x})$' is the *prototypical* definition of $A(\mathbf{x})$.

Let '$X(\mathbf{x}) \Rightarrow \Delta X(\mathbf{x})$' be the rule obtained from (2) by applying Clark's Predicate Completion and then replacing the predicate constant A with the predicate variable X. We say that '$X(\mathbf{x}) \Rightarrow \Delta X(\mathbf{x})$' is the *transformation* associated with $A(\mathbf{x})$. □

Now let $\Phi(A)$ be any Horn clause in which the predicate constant A appears on the right-hand side. For example:

$$\Phi(A) \equiv (\forall \mathbf{x}) \left[P(\mathbf{x}) \Leftarrow A(\mathbf{x}) \wedge \bigwedge_{i=1}^{l} B_i(\mathbf{x}) \right]$$

We treat $\Phi(A)$ as a schema that depends on A, so that we are free to substitute A^0, ΔX and X as we wish.

Definition 2.5: The *induction schema* for $\Phi(A)$ is the following sentence in second-order intuitionistic logic:

$$\Phi(A) \Leftarrow \Phi(A^0) \wedge (\forall X)[\Phi(\Delta X) \Leftarrow \Phi(X)]$$

The interesting point about this induction schema is that it takes the form of an embedded implication with an embedded second-order universal quantifier. Second-order intuitionistic logic has no complete proof procedure, of course, but it turns out that a set of second-order sentences in this form *does* have a complete proof procedure. The procedure is similar to the first-order proof procedure for universally quantified implications discussed in [18]. To prove the second conjunct on the right-hand side of Definition 2.5, we replace the predicate variable 'X' with a new predicate constant '!X', we assert $\Phi(!X)$ into the rulebase, and we try to prove $\Phi(\Delta!X)$. If this proof succeeds, then we have proven the goal: $(\forall X)[\Phi(\Delta X) \Leftarrow \Phi(X)]$. For a proof that this procedure is complete, see [20].

We will show how to use this induction schema in Sections 3 and 4. The justification of our approach is given in the following two theorems, which are proven in [20] using the concept of a final Kripke model. In the statement of these theorems, $\boldsymbol{A}$ is a tuple consisting of the recursively defined predicates in $\mathcal{R}$, which is assumed to include only linear recursive definitions, $\mathcal{P}(\boldsymbol{A})$ denotes the set of prototypical definitions of the predicates in $\boldsymbol{A}$ given by Definition 2.4, and $\mathcal{S}(\boldsymbol{A})$ denotes the set of all induction schemata for the predicates in $\boldsymbol{A}$ that can be constructed using Definition 2.5.

Theorem 2.6: $\mathcal{Q} \cup \mathit{Comp}(\mathcal{R}) \cup \mathcal{P}(\boldsymbol{A}) \models \psi \iff \mathcal{Q} \cup \mathit{Circ}(\mathcal{R}(\boldsymbol{P}); \boldsymbol{P}) \models \psi$

Theorem 2.7: $\mathcal{Q} \cup \mathit{Comp}(\mathcal{R}) \cup \mathcal{S}(\boldsymbol{A}) \models \psi \implies \mathcal{Q} \cup \mathit{Circ}(\mathcal{R}(\boldsymbol{P}); \boldsymbol{P}) \models \psi$

These theorems suggest that we search first for a *prototypical proof* of ψ, i.e., a proof that uses just the prototypical definitions $\mathcal{P}(\boldsymbol{A})$. If we fail to find a prototypical proof, we have failed, period. But if we succeed, we can analyze the successful prototypical proof in an attempt to construct an induction schema in $\mathcal{S}(\boldsymbol{A})$. We can then search for a proof from this induction schema, using the procedure for second-order embedded implications outlined above.

Intuitively, Theorem 2.6 tells us that prototypical proofs are complete but not necessarily sound, while Theorem 2.7 tells us that inductive proofs are sound but not necessarily complete. We will see how to combine these two proof procedures in the following two sections of the paper.

3 Example: Red and Green Blocks

The example in this section is taken from [21]. Let $\mathcal{R}$ be the following set of rules:

$$\text{ChristmasBlock}(x) \Leftarrow \text{Block}(x) \wedge \text{Red}(x) \tag{1}$$

$$\text{ChristmasBlock}(x) \Leftarrow \text{Block}(x) \wedge \text{Green}(x) \tag{2}$$

$$\text{OnCB}(x, y) \Leftarrow \text{ChristmasBlock}(x) \wedge \text{ChristmasBlock}(y) \wedge \text{On}(x, y) \tag{3}$$

$$\text{AboveCB}(x, y) \Leftarrow \text{OnCB}(x, y) \tag{4}$$

$$\text{AboveCB}(x, y) \Leftarrow \text{OnCB}(x, z) \wedge \text{AboveCB}(z, y) \tag{5}$$

Intuitively, rules (1)–(2) define the concept of a 'ChristmasBlock', and rules (3)–(5) define the concept of a stack of 'ChristmasBlocks'. Suppose we are told that there exists a stack of 'ChristmasBlocks' in which block 'a' is above block 'b', and furthermore that 'a' and 'b' are painted green and red, respectively. Does it follow that there is something green on something red?

Intuitively, the answer should be: Yes. Formally, we can pose this question by circumscribing the predicates 'ChristmasBlock,' 'OnCB' and 'AboveCB' in rules (1)–(5), adding the following Horn clause to $\mathcal{Q}$:

$$\text{GreenOnRed} \Leftarrow \text{On}(x, y) \wedge \text{Green}(x) \wedge \text{Red}(y), \tag{6}$$

and taking ψ to be the following implication:

$$\text{GreenOnRed} \Leftarrow \text{AboveCB}(\text{a}, \text{b}) \wedge \text{Green}(\text{a}) \wedge \text{Red}(\text{b}). \tag{7}$$

We now try to show that $\mathcal{Q} \cup \mathit{Circ}(\mathcal{R}(\boldsymbol{P});\boldsymbol{P}) \models \psi$.

A successful proof is shown in Figures 1 and 2. Rules (4)–(5) constitute a linear recursive definition of the predicate 'AboveCB', in which

$$\text{AboveCB}(x, y) \Rightarrow \text{OnCB}(x, y) \tag{8}$$

is the prototypical definition, and

$$X(x, y) \Rightarrow (\exists z)[\text{OnCB}(x, z) \wedge X(z, y)] \tag{9}$$

is the transformation. Using the notation in Definition 2.4, the right-hand side of (8) is written as '$\text{AboveCB}^0(x, y)$', and the right-hand side of (9) is written as '$\Delta X(x, y)$'. Since we are trying to prove the implication in (7), we construct an initial tableau, $\mathcal{T}_0$, with 'AboveCB(a,b)', 'Green(a)' and 'Red(b)' in its data base, and with 'GreenOnRed' as its goal, and we try to show that this goal succeeds using the prototypical definition in (8). Figure 1 shows a successful proof, which happens

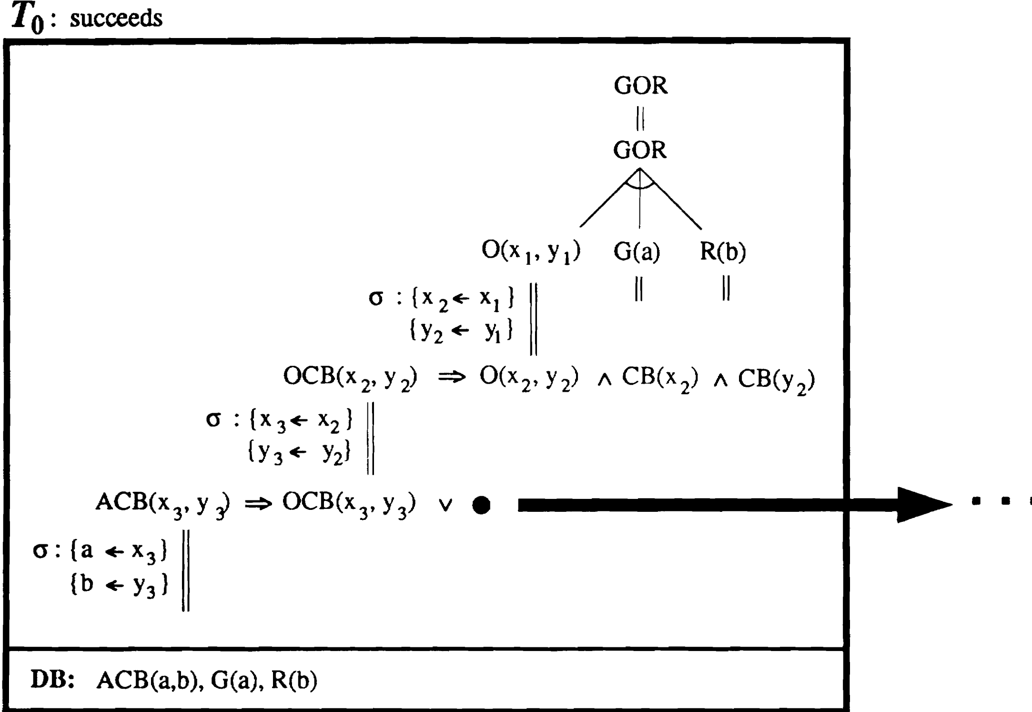


Figure 1: "Red and Green Blocks," prototypical proof.

to use $Comp(\mathcal{R})$ applied to rule (3). We have thus found a prototypical proof, as guaranteed by Theorem 2.6.

Our task now is to "strengthen" the proof from $\mathcal{P}(\boldsymbol{A})$ into a proof from $\mathcal{S}(\boldsymbol{A})$, if possible. The first step is to generalize the proof in Figure 1 from a proof that works for the constant 'a' to a proof that works for the variable 'x'. (See [12] for the analysis of a similar problem in "explanation-based generalization".) It is easy to see that this generalization is successful. We now have a proof of the following universally quantified implication:

$$(\forall x)[\text{GreenOnRed} \Leftarrow \text{OnCB}(x,\text{b}) \wedge \text{Green}(x) \wedge \text{Red}(\text{b})]. \tag{10}$$

Let us call this implication $\Phi(\text{AboveCB}^0)$. Then $\Phi(\text{AboveCB})$ is the following universally quantified implication:

$$(\forall x)[\text{GreenOnRed} \Leftarrow \text{AboveCB}(x,\text{b}) \wedge \text{Green}(x) \wedge \text{Red}(\text{b})]. \tag{11}$$

If we can prove (11), we will also have a proof of our original query (7). Therefore, using the induction schema in Definition 2.5, we try to prove $(\forall X)[\Phi(\Delta X) \Leftarrow \Phi(X)]$. This goal is an implication with a second-order universal quantifier, so we create a new tableau, $\mathcal{T}_1$, we add $\Phi(!\text{X})$ to the data base, and we try to prove $\Phi(\Delta!\text{X})$ in $\mathcal{T}_1$.

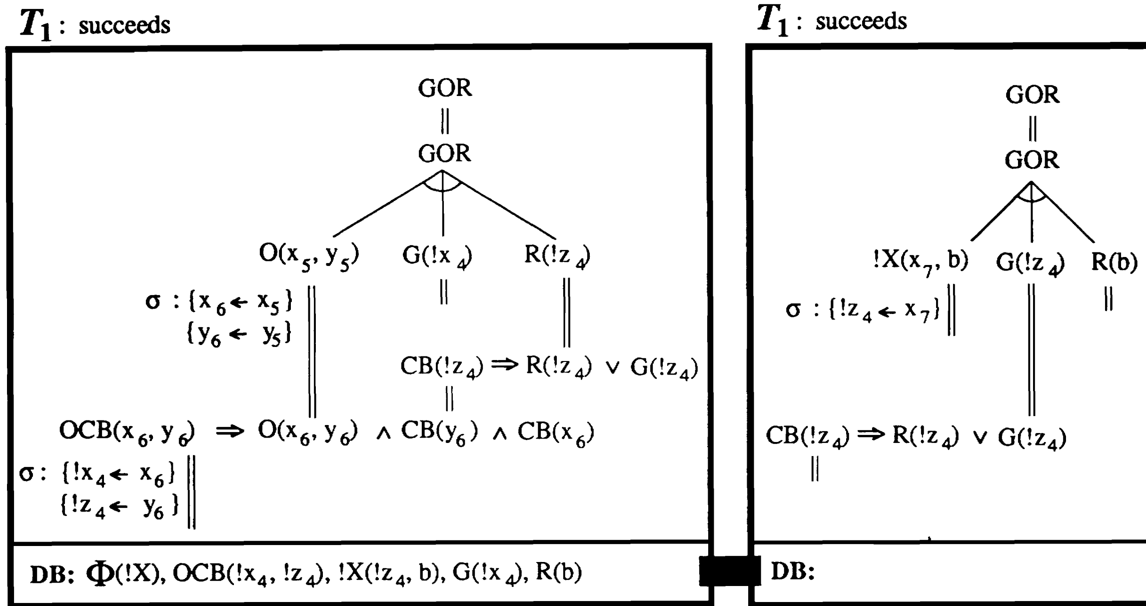


Figure 2: "Red and Green Blocks," inductive proof.

Let us write out each of these schemata in detail. $\Phi(!X)$ is the following implication:

$$(\forall x)[\text{GreenOnRed} \Leftarrow !X(x, b) \wedge \text{Green}(x) \wedge \text{Red}(b)], \tag{12}$$

and $\Phi(\Delta!X)$ is equivalent to the following implication:

$$(\forall x, z)[\text{GreenOnRed} \Leftarrow \text{OnCB}(x, z) \wedge !X(z, b) \wedge \text{Green}(x) \wedge \text{Red}(b)]. \tag{13}$$

To prove (13), we instantiate 'x' and 'z' to the special constants '$!x_4$' and '$!z_4$', we add the right-hand side of (13) to the data base of $\mathcal{T}_1$, and we try to prove the left-hand side of (13). The proof is shown in Figure 2. The main point to note is that the proof now uses $Comp(\mathcal{R})$ applied to rules (1) and (2), which generates a disjunctive assertion. It is therefore necessary to use a "disjunctive splitting" operation [14] in order to obtain a closed proof. However, Figure 2 shows that the goal 'GreenOnRed' succeeds initially from the disjunct 'Red($!z_4$)', and then succeeds again from the disjunct 'Green($!z_4$)' using $\Phi(!X)$.

We have thus shown, by Theorem 2.7, that $\mathcal{Q} \cup Circ(\mathcal{R}(\boldsymbol{P});\boldsymbol{P}) \models \psi$.

4 Example: Naive Reverse

The problem in Section 3 is relatively simple, but we have constructed proofs of this sort for more complicated problems. In particular, we have applied our techniques to prove various properties of

PROLOG programs [10, 3]. For example, let 'Append(l,m,n)' be defined as usual:

$$\text{Append}(\text{nil}, l, l) \Leftarrow \tag{14}$$

$$\text{Append}([k \mid l], m, [k \mid n]) \Leftarrow \text{Append}(l, m, n) \tag{15}$$

Let 'Reverse(r,s)' be defined as follows:

$$\text{Reverse}(\text{nil}, \text{nil}) \Leftarrow \tag{16}$$

$$\text{Reverse}([q \mid r], p) \Leftarrow \text{Reverse}(r, s) \wedge \text{Append}(s, [q], p) \tag{17}$$

Intuitively, 'Reverse' should be a symmetric relation. We can express this property by taking ψ to be the following universally quantified implication:

$$(\forall x, y)[\text{Reverse}(y, x) \Leftarrow \text{Reverse}(x, y)]. \tag{18}$$

We now show that (18) is entailed by the circumscription of 'Append' and 'Reverse' in rules (14)–(17).

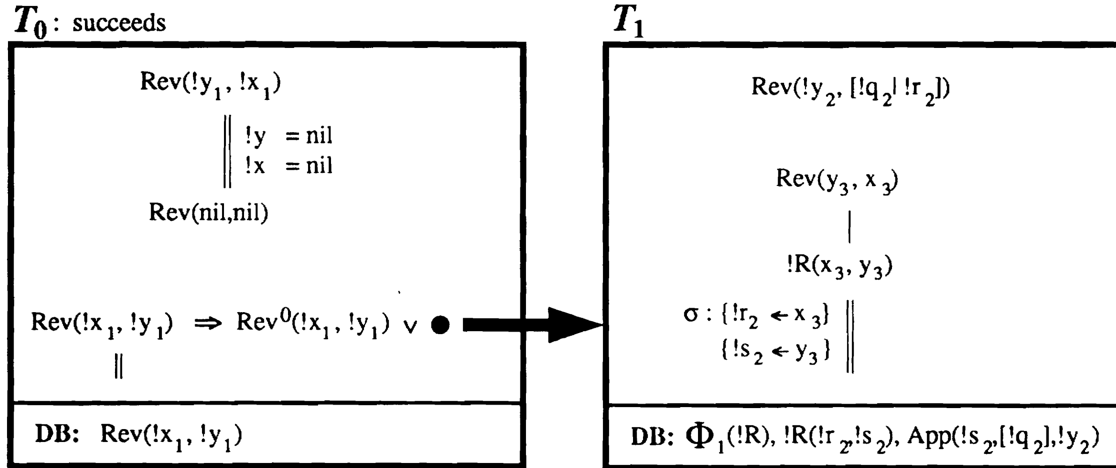


Figure 3: "Naive Reverse," partial proof.

The first part of the proof is shown in Figure 3. Applying our first-order proof procedure for intuitionistic embedded implications [18], we construct an initial tableau, $\mathcal{T}_0$, with 'Reverse($!x_1, !y_1$)' in its data base and with 'Reverse($!y_1, !x_1$)' as its goal. The prototypical definition of 'Reverse' is given by Definition 2.4, as before, but its use in the tableau proof is slightly more complicated here than it was in Section 3. Applying Clark's Predicate Completion to rule (16) alone, we have:

$$\text{Reverse}(x, y) \Rightarrow x = \text{nil} \ \wedge \ y = \text{nil}. \tag{19}$$

Thus 'Reverse^0(!x$_1$, !y$_1$)' is the assertion that '$!x_1 = \text{nil}$' and '$!y_1 = \text{nil}$', and when these values are substituted throughout the tableau $\mathcal{T}_0$ the goal succeeds immediately, as indicated in Figure 3. We thus have a proof of the following universally quantified implication:

$$(\forall x, y)[\text{Reverse}(y, x) \Leftarrow \text{Reverse}^0(x, y)]. \tag{20}$$

Let us call this implication $\Phi_1(\text{Reverse}^0)$. Then the implication in (18), our ultimate goal, is $\Phi_1(\text{Reverse})$.

The prototypical proof in Figure 3 has suggested an induction schema, and we now compute the expression $(\forall R)[\Phi_1(\Delta R) \Leftarrow \Phi_1(R)]$ where R is a predicate variable. We can immediately write:

$$\Phi_1(R) \;\equiv\; (\forall x, y)[\text{Reverse}(y, x) \Leftarrow R(x, y)]. \tag{21}$$

Also, by Definition 2.4, the transformation associated with 'Reverse' is:

$$R(x, y) \;\Rightarrow\; (\exists q, r, s) \qquad R(r, s) \;\wedge\; \text{Append}(s, [q], y) \;\wedge\; x = [q \mid r], \tag{22}$$

and we can therefore write:

$$\Phi_1(\Delta R) \;\equiv\; (\forall x, y, q, r, s) \qquad \text{Reverse}(y, x) \Leftarrow R(r, s) \;\wedge\; \text{Append}(s, [q], y) \;\wedge\; x = [q \mid r] \tag{23}$$

Tableau $\mathcal{T}_1$ in Figure 3 shows our attempt to prove the right-hand side of this induction schema. We add $\Phi_1(!\text{R})$ to the data base and we try to prove $\Phi_1(\Delta!\text{R})$. Notice that the equality '$x = [q \mid r]$' in (23) can be eliminated when we attempt this proof.

However, as Figure 3 indicates, this proof does not succeed immediately. Instead, we are able to reduce the goal in tableau $\mathcal{T}_1$ to another universally quantified implication:

$$(\forall y, q, r, s)[\text{Reverse}(y, [q \mid r]) \Leftarrow \text{Reverse}(s, r) \;\wedge\; \text{Append}(s, [q], y)]. \tag{24}$$

We now attempt, in Figure 4, to prove (24). The strategy here is exactly the same: Find a proof using the prototypical definitions $\mathcal{P}(\boldsymbol{A})$, and then try to "strengthen" this proof into a proof from $\mathcal{S}(\boldsymbol{A})$. The prototypical definition of 'Append' is:

$$\text{Append}(x, y, z) \Rightarrow y = z \;\wedge\; x = \text{nil}. \tag{25}$$

Thus, to assert 'Reverse^0(!s$_2$, !r$_2$)' and 'Append^0(!s$_2$, [!q$_2$], !y$_2$)' is to assert '$!s_2 = !r_2 = \text{nil}$' and '$!y_2 = [!q_2]$'. When these values are substituted throughout the tableau $\mathcal{T}_1$, as shown in Figure 4, the goal succeeds. We thus have a proof of the following universally quantified implication:

$$(\forall y, q, r, s)[\text{Reverse}(y, [q \mid r]) \Leftarrow \text{Reverse}^0(s, r) \;\wedge\; \text{Append}^0(s, [q], y)]. \tag{26}$$

Our task now is to strengthen the proof of (26) into a proof of (24).

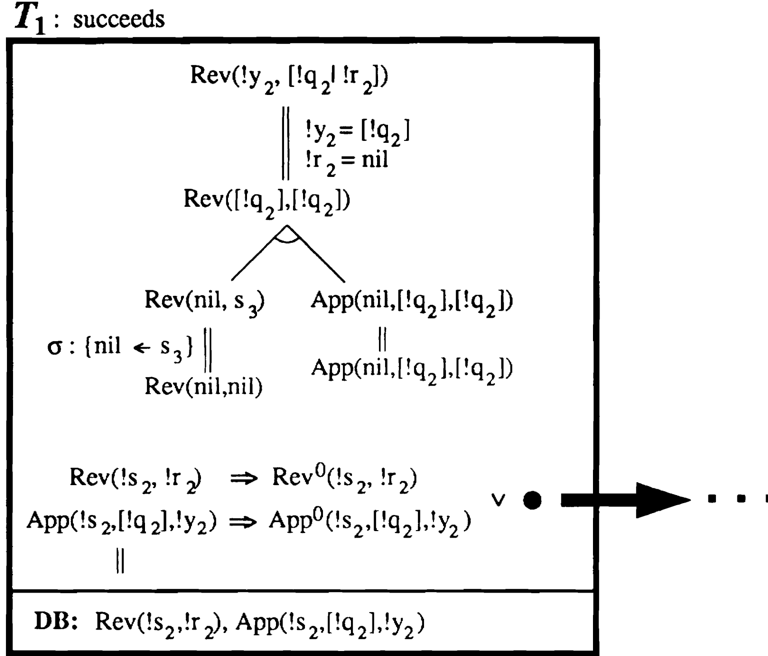


Figure 4: "Naive Reverse," second prototypical proof.

Since there are two recursive predicates on the right-hand side of (24), we can expect the construction of an induction schema here to be more complicated than it was in our prior examples. However, it turns out that we can transform the relations 'Reverse' and 'Append' *conjunctively* in this case. (In other cases, alternative strategies may be necessary.) Suppose we define:

$$\Phi_2(R \wedge A) \equiv (\forall y, q, r, s)[\text{Reverse}(y, [q \mid r]) \Leftarrow R(s, r) \wedge A(s, [q], y)], \tag{27}$$

where R and A are predicate variables and '$R \wedge A$' is their conjunction. By Definition 2.4, the transformation for 'Append' is:

$$A(x, y, z) \Rightarrow (\exists k, l, n) \quad A(l, y, n) \wedge x = [k \mid l] \wedge z = [k \mid n], \tag{28}$$

and combining this with the transformation for 'Reverse', we have:

$$R(s, r) \wedge A(s, [q], y) \Rightarrow (\exists k, l, n, z) \quad R(l, z) \wedge \text{Append}(z, [k], r) \wedge A(l, [q], n) \wedge s = [k \mid l] \wedge y = [k \mid n]. \tag{29}$$

Notice, because of the equality '$s = [k \mid l]$' in (29), that the first arguments of R and A will always be identical under the application of this transformation. This is the key observation that allows us to compose the transformations (22) and (28) conjunctively in this case, and it is also the property

that allows the "merger" of the induction schemata in [10]. Finally, substituting the right-hand side of (29) into the schema Φ_2, we have:

$$\Phi_2(\Delta R \wedge A) \equiv (\forall y, q, r, k, l, n, z) \quad \text{Reverse}(y, [q \mid r]) \Leftarrow R(l, z) \wedge \text{Append}(z, [k], r) \wedge A(l, [q], n) \wedge y = [k \mid n]. \tag{30}$$

Figure 5 now shows that the proof using this induction schema is successful.

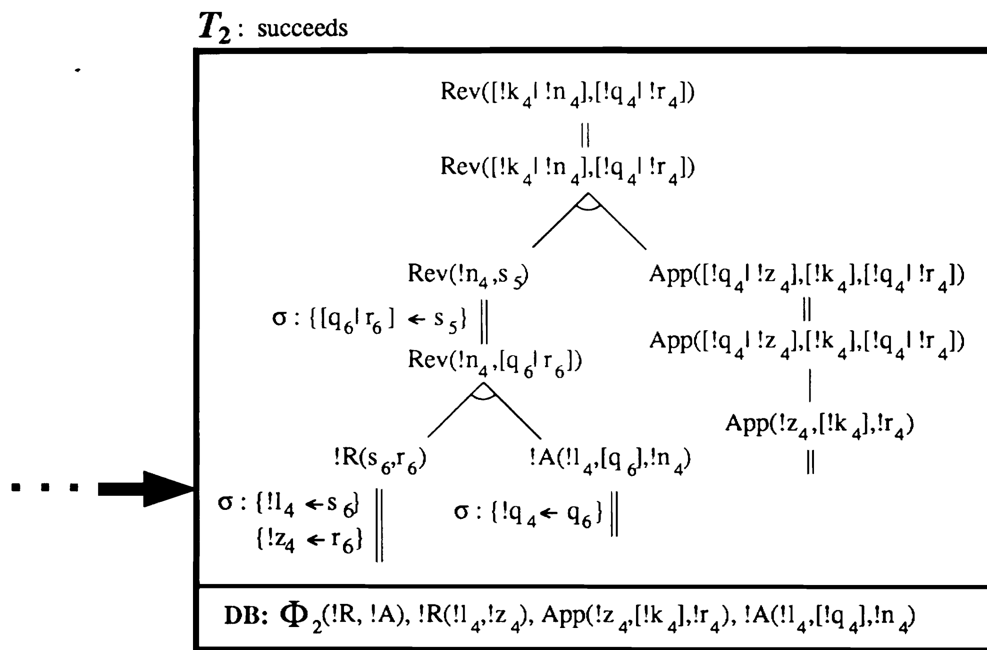


Figure 5: "Naive Reverse," second inductive proof.

We have thus shown, by Theorem 2.7, that (18) is entailed by the circumscription of 'Append' and 'Reverse' in rules (14)–(17).

5 Current Work

This work is currently being extended in two directions:

1. We are analyzing a wider class of recursive definitions.
2. We are writing a PROLOG interpreter to search for inductive proofs.

Preliminary results of these investigations will be reported at the workshop.

References

[1] K.L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum, 1978.

[2] K.L. Clark and S.-A. Tarnlund. A first-order theory of data and programs. In B. Gilchrist, editor, *Information Processing '77 (IFIP Proceedings)*, pages 939–944. North-Holland, 1977.

[3] C. Elkan and D. McAllester. Automated inductive reasoning about logic programs. In *Proceedings, Fifth International Conference and Symposium on Logic Programming*, pages 876–892, 1988.

[4] L. Fribourg. Equivalence-preserving transformations of inductive properties of PROLOG programs. In *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, pages 893–908. MIT Press, 1988.

[5] L. Fribourg. Extracting logic programs from proofs that use extended PROLOG execution and induction. In *Logic Programming: Proceedings of the Seventh International Conference*, pages 685–699. MIT Press, 1990.

[6] L. Fribourg. Automatic generation of simplification lemmas for inductive proofs. In *Proceedings, 1991 International Logic Programming Symposium*, pages –. MIT Press, 1991.

[7] D.M. Gabbay. N-PROLOG: An extension of PROLOG with hypothetical implication. II. Logical foundations, and negation as failure. *Journal of Logic Programming*, 2:251–283, 1985.

[8] D.M. Gabbay and U. Reyle. N-PROLOG: An extension of PROLOG with hypothetical implications. I. *Journal of Logic Programming*, 1:319–355, 1984.

[9] J. Hsiang and M. Srivas. Automatic inductive theorem-proving using PROLOG. *Theoretical Computer Science*, 54:3–28, 1987.

[10] T. Kanamori and H. Fujita. Formulation of induction formulas in verification of PROLOG programs. In *Proceedings, Eighth International Conference on Automated Deduction*, pages 281–299, 1986.

[11] T. Kanamori and H. Seki. Verification of PROLOG programs using an extension of execution. In *Proceedings, Third International Conference on Logic Programming*, pages 475–589, 1986.

[12] S. Kedar-Cabelli and L.T. McCarty. Explanation-based generalization as resolution theorem proving. In *Proceedings of the Fourth International Workshop on Machine Learning*, pages 383–389. Morgan Kaufmann, 1987.

[13] V. Lifschitz. Computing circumscription. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 121–127, 1985.

[14] D.W. Loveland. Near-Horn Prolog and beyond. Technical Report CS-1988-25, Department of Computer Science, Duke University, 1988. To appear in *Journal of Automated Reasoning*.

[15] J. McCarthy. Circumscription: A form of non-monotonic reasoning. *Artificial Intelligence*, 13:27–39, 1980.

[16] J. McCarthy. Applications of circumscription to formalizing common-sense knowledge. *Artificial Intelligence*, 28:89–116, 1986.

[17] L.T. McCarty. Clausal intuitionistic logic. I. Fixed-point semantics. *Journal of Logic Programming*, 5(1):1–31, 1988.

[18] L.T. McCarty. Clausal intuitionistic logic. II. Tableau proof procedures. *Journal of Logic Programming*, 5(2):93–132, 1988.

[19] L.T. McCarty. Circumscribing embedded implications. In A. Nerode et al., editors, *Proceedings, First International Workshop on Logic Programming and Non-Monotonic Reasoning*, pages 211–227. MIT Press, 1991.

[20] L.T. McCarty. Computing with prototypes. Technical report, Computer Science Department, Rutgers University, 1992. Submitted for publication.

[21] L.T. McCarty and R. van der Meyden. Indefinite reasoning with definite rules. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, pages 890–896, 1991.

[22] L.T. McCarty and R. van der Meyden. Reasoning about indefinite actions. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference (KR92)*, page (forthcoming). Morgan Kaufmann, October 1992.

[23] D. Miller. A logical analysis of modules in logic programming. *Journal of Logic Programming*, 6:79–108, 1989.

[24] D. Miller, G. Nadathur, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[25] G. Nadathur and D.A. Miller. An overview of λPROLOG. In *Proceedings, Fifth International Conference and Symposium on Logic Programming*, pages 810–827, 1988.

[26] R. Reiter. Circumscription implies predicate completion (sometimes). In *Proceedings of the Second National Conference on Artificial Intelligence*, pages 418–420, 1982.

[27] A. Troelstra and D. van Dalen. *Constructivism in Mathematics: An Introduction*. North-Holland, 1988.

An Empirical Study of the Runtime Behavior of Higher-Order Logic Programs [1]

(Preliminary Version)

Spiro Michaylov
Department of Computer and Information Science
The Ohio State University
228 Bolz Hall
2036 Neil Avenue Mall
Columbus, OH 43210-1277, U.S.A.
spiro@cis.ohio-state.edu

Frank Pfenning
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890, U.S.A.
fp@cs.cmu.edu

1 Introduction

Implementation technology for higher-order logic programming languages such as λProlog [17] and Elf [21] is still in its infancy. There are many features of these languages that do not occur in ordinary Prolog programs, such as types, variable binding constructs for terms, embedded implication and universal quantification, or dependent types and explicit construction of proofs. Some initial work on compiler design for higher-order logic programming languages can be found in [11, 16, 18, 19][2]. At the same time, the language design process for such languages is far from complete. Extensions [2, 7] as well as restrictions [14] of λProlog have been proposed to increase its expressive power or simplify the language theory or its implementation.

Obviously, further language design and implementation efforts must be closely linked. It is easy to design unimplementable languages or implement unusable languages. In order to understand and evaluate the challenges and available choices, we report the results of an empirical study of existing example programs. We chose Elf over λProlog for this study for two reasons: (1) accessibility of the large suite of examples, and (2) ease of instrumentation of the Elf interpreter to perform measurements. Many of these examples can be trivially transformed into λProlog programs, and essentially the same issues arise regarding their runtime behavior. We will discuss later which

[1]This research was sponsored partly by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

[2]See also the paper by Kwon and Nadathur in this volume

measurements are specific to Elf.

Currently, we have access to about 10,000 lines of Elf code, written mostly by the authors and students in a course on *Computation and Deduction* taught in the Spring of 1992. We selected a sample of 12 representative examples of about 3500 total lines of code to conduct this study. The examples cover a range of applications from logic and the theory of programming languages. They are explained further in Section 3.

We briefly summarize what we consider to be some of the central issues and our conclusion.

Full unification in higher-order languages is clearly impractical, due to the non-existence of minimal complete sets of most-general unifiers [8]. Therefore, work on λProlog has used Huet's algorithm for *pre-unification* [8], where so-called flex-flex pairs (which are always unifiable) are postponed as constraints, in effect turning λProlog into a constraint logic programming language. Yet, even pre-unifiability is undecidable, and sets of most general pre-unifiers may be infinite. While undecidability has not turned out to be a severe problem, the lack of unique most general unifiers makes it difficult to accurately predict the run-time behavior of a λProlog program that attempts to take advantage of full higher-order pre-unification. It can result in thrashing when certain combinations of unification problems have to be solved by extensive backtracking. Moreover, in a straightforward implementation, common cases of unification incur a high overhead. These problems have led to a search for natural, decidable subcase of higher-order unification. Miller [14] has suggested a syntactic restriction (L_λ) to λProlog, easily extensible to related languages [22], where most general unifiers are unique modulo $\beta\eta\alpha$-equivalence.

Miller's restriction has many attractive features. Unification is deterministic and thrashing behavior due to unification is avoided. Higher-order unification in its full power can be implemented if some additional control constructs (**when**) are available [15].

However, our study suggests that this solution is unsatisfactory, since it has a detrimental effect on programming methodology, and potentially introduces a new efficiency problem. Object-level variables are typically represented by meta-level variables, which means that object-level capture-avoiding substitution can be implemented via meta-level β-reduction. The syntactic restriction to L_λ prohibits this implementation technique, and hence a new substitution predicate must be programmed for each object language. Not only does this make programs harder to read and reason about, but a substitution predicate will be less efficient than meta-language substitution.

This is not to diminish the contribution that L_λ has made to our understanding of higher-order logic programming. The operational semantics of Elf, in contrast to λProlog, is based on solving all dynamically arising equations that lie within an appropriate extension of L_λ to dependent types. All other equations (solvable or not) are postponed as constraints. We found that this addresses the problems with higher-order unification without compromising programming methodology.

This still leaves open the question whether this constraint satisfaction algorithm can be implemented efficiently. Part of our study was aimed at determining the relative frequency of various forms of equations, in order to guide future design of efficient implementations.

In this paper we study the run-time behavior of a large suite of Elf programs, and demonstrate the following:

- While a large proportion of programs are outside L_λ syntactically, the cases of unification that occur dynamically are almost all deterministic.

- All of the programs behave well if nondeterministic cases of unification are delayed until they are deterministic.
- While most programs at some point use non-trivial cases of higher-order unification, the vast majority of unification instances are extremely simple, in fact, essentially Prolog unification.

This empirical study has been performed by instrumenting an Elf interpreter to count:

- the relative frequency of different cases of unification,
- the relative frequency of various instances of substitution,
- the number of times non-deterministic unification would arise were these cases not delayed.

This leads us to suggest a strategy for efficient implementation of higher-order logic programming languages, which is essentially the strategy described for Constraint Logic Programming languages in [9, 12]. That is:

- The languages should not be restricted syntactically.
- The unification instances corresponding to those of L_λ should be identified as *directly solvable*, and the remainder as *hard*. Hard constraints should be delayed until they become directly solvable as a result of further variable instantiation. The relevant terminology, concepts and implementation methods are described in [10].
- Data structures and algorithms should be designed to favor the simple cases of unification.

2 Properties of Programs

Since our concern in this paper is with efficient implementation (and its interaction with language design), the properties of programs that we most need to study are the dynamic properties: how frequently do various phenomena arise when typical queries are executed? This allows us to tune data structures and algorithms. On the other hand, to evaluate the possibility of syntactic restrictions, we also need to know what occurs syntactically in programs. We begin by discussing these syntactic properties and why they are of interest. Then we go on to discuss the dynamic properties.

2.1 Static Properties

L_λ vs. general variable applications. Because of our interest in the syntactic restriction to L_λ, we need to understand how often and why programs do not fall into this subset. An important use of general variable applications appears in a rule like the following (taken from a natural semantics in [13])

```
eval_app_lam    : eval (app E1 E2) V
                        <- eval E1 (lam E1')
                        <- eval E2 V2
                        <- eval (E1' V2) V.
```

where we see an application of two existential variables (E1' V2) to implement substitution in an object language by meta-level β-reduction.

Even within the L_λ subset, we can observe interesting static properties of programs. For example, many programs structurally recurse through an object language expression, where the object is represented using higher-order abstract syntax. Consider the rule above: the head of this rule requires only first order unification, which could be implemented as simple variable binding.

Type redundancy. Both in λProlog and Elf there is a potential for much redundant runtime type computation. In λProlog, this is due to polymorphism (see [11]), in Elf it is due to type dependency. Such redundancy can be detected statically. However, the question about the dynamic properties of programs remains: how much type computation remains after all redundant ones have been eliminated.

Level of indexing. This is an Elf-specific property of a program. Briefly, a (simple) type is a level 0 type family. A type family indexed by objects of level 0 type is a level 1 type family. In general, an type family indexed by objects whose type involves level n families is a family of level $n+1$. For example,

```
o : type.                      % propositions, level 0.
pf : o -> type.                % proofs of propositions, level 1.
norm : pf A -> pf A -> type.   % proof transformations, level 2.
proper : norm P Q -> type.     % proper proof transformations, level 3.
```

This is of interest because the level of indexing determines the amount of potentially redundant type computation. Empirically, it can be observed that programs at level 2 or 3 have in some respects different runtime characteristics than programs at level 1. We have therefore separated out the queries of the higher-level. This also helps to separate out the part of our analysis which is directly relevant to λProlog, where all computation happens at levels 0 and 1 (due to the absence of dependency).

2.2 Dynamic Properties

The major dynamic properties studied in this paper are substitution, unification and constraint solving.

Substitution. Substitution can be a significant factor limiting performance. It is thus important to analyze various forms of substitution that arise during execution. When measuring these, our concern is simple: substitutions with anything other than parameters (*uvars*) result from the fragment of the language outside L_λ, so these represent substitutions that would have had to have been performed using Elf code if the L_λ restriction had been applied. Moreover, the relative frequency of parameter substitution suggests that it is crucial for it to be highly efficient, while general substitution is somewhat less critical. A proposal regarding efficient implementation of terms has been made in [18]. For our study we eliminated substitutions which arose due to clause copying and during type reconstruction, since these are residuals effects of the interpreter and would most likely be eliminated in any reasonable compiler.

Unification and Constraint Satisfaction. We measure various aspects of unification and constraint satisfaction. Terms involved in equations (disagreement pairs) are classified as *rigid* (con-

stant head), *uvars* (parameters, *i.e.*, temporary constants), *evars* (simple logic variables), *gvars* (generalized variables, *i.e.*, logic variables applied to distinct, dominated parameters [14]), *flexible* (compound terms with a logic variable at the head, but not a gvar), *abst* (a term beginning with a λ-abstraction), or *quant* (a term beginning with a Π-quantification, in Elf only).

One of our goals is to determine how close Elf computations come to Prolog computations in several respects:

- How many pairs, at least at the top level, require essentially Herbrand unification? These are the rigid-rigid and evar-anything cases.
- How many pairs still have unique mgus, that is, gvar-gvar, or admit a unique strategy for constraint simplification, that is, gvar-rigid, abst-anything, or quant-anything?
- How often do the remaining cases arise (which are postponed to avoid branching)?
- How successful is rule indexing (as familiar from Prolog) to avoid calls to unification?

In our opinion, while we have not yet completed the required experiments, it is also very important to determine the following:

- How important is the occurs-check (extended to deal with a dependency check)?
- How much time is spent on type computations as compared to object computations?
- How much time is spent on proof computations, when it is requested by the user or required for further computation?

3 Study of Programs

In this section we report our preliminary findings. We currently have detailed statistics on the kinds of disagreement pairs that arise during unification, and the kind of substitution that is performed during unification and search.

3.1 The Examples

Figures 1 and 2 show the data for basic computation queries and proof manipulation queries respectively, for the range of programs. Thus Figure 1 is especially applicable to the understanding of λProlog programs, while Figure 2 measures Elf-specific behavior.

The two tables in each figure give data on five areas of interest, as follows:

- *All Unifications*
  The total gives an indication of computational content, while the breakdown indicates the usefulness of first-argument indexing and the amount of deep search.

  Unif Total number of subgoal/head pairs to be unified.
  %Ind Percentage of above total unifications avoided by rule indexing.
  %S Percentage of total unifications that succeeded.
  %F Percentage of total unifications that failed.

- *Dynamic Unifications*
  It is also useful to have this information for rules assumed through embedded implication, since indexing of such rules is more complicated, and compilation has a runtime cost.

  Dyn Total number of subgoal/head pairs to be unified, where the head is from a rule assumed (dynamically) through embedded implication.
  %Ind, *%S*, *%F*
  Percentages of number of unifications with heads from dynamic rules, as above.

- *Dynamic/Assume*
  By knowing how many rules are assumed dynamically, and on average how often they are used, we can see whether it is worthwhile to index and compile such rules or whether they should be interpreted.

  Ass Number of rules assumed by implication.
  U/Ass Normalized ratio of total unifications with dynamic rules to number of rules assumed by implication.
  AU/Ass As above, but using only those rules where the unification was not avoided through indexing.

- *Disagreement Pairs*
  We study the kinds of disagreement pairs that arise to determine which kinds of unification dominate.

  Tot Total number of disagreement pairs examined throughout the computation.
  %E-? Percentage of disagreement pairs that involved a simple evar.
  %G-? Percentage of disagreement pairs that involved a gvar which is *not* a simple evar.
  %R Percentage of disagreement pairs between two rigid terms.
  %A Percentage of disagreement pairs between two abstractions.

- *Substitutions*
  Substitutions and abstractions (the inverse of uvar substitutions) are expensive, and the efficiency of one can be improved at the expense of the other. Furthermore, some kinds of substitutions are more costly than others. Thus it is useful to know what kinds of substitutions arise, how often both substitution and abstraction arise, and their relative frequency.

  Tot Total number of substitutions for bound variables.
  %Uv Percentage of the above where a uvar is substituted.
  Abs Number of abstractions over a uvar.
  Abs/Uv Normalized ratio of such abstractions to substitutions of uvars.

The examples used are as follows:

- Extraction — *Constructive theorem proving and program extraction* [1]

Program	All Unifications				Dynamic Unifications				Dynamic/Assume		
	Unif	%Ind	%S	%F	Dyn	%Ind	%S	%F	Ass	U/Ass	AU/Ass
Mini-ML	15333	87	13	0	1532	93	7	0	67	22.87	1.61
Canonical	177	66	28	6	8	50	50	0	3	2.67	1.33
Prop	677	60	30	10	41	44	41	15	9	4.56	2.56
F-O	359	65	28	7	33	18	82	0	17	1.94	0.07
Forsythe	2087	38	23	39	16	25	75	0	10	1.60	1.20
Lam	240	50	40	10	26	80	15	5	4	6.50	1.25
Polylam	982	65	34	1	389	88	12	1	45	8.64	1.00
Records	2459	61	31	8	274	61	39	0	28	9.79	3.79
DeBruijn	451	25	39	36	5	40	60	0	5	1.00	0.60
CLS	278	0	32	68	0	-	-	-	0	-	-

Program	Disagreement Pairs					Substitutions			
	Tot	%E-?	%G-?	%R	%A	Tot	%Uv	Abs	Abs/Uv
Mini-ML	8716	47	0	52	0	6411	98	0	0.00
Canonical	427	41	8	56	0	180	96	36	0.21
Prop	1681	54	0	45	1	202	100	8	0.04
F-O	438	40	6	58	0	108	100	58	0.54
Forsythe	5812	43	0	57	0	39	100	0	0.00
Lam	874	41	0	59	0	149	86	0	0.00
Polylam	2085	48	3	50	1	7907	89	81	0.01
Records	3880	46	3	53	0	1347	100	204	0.15
DeBruijn	1554	44	1	56	0	688	97	16	0.02
CLS	2455	36	0	65	0	0	-	0	-

Figure 1: Basic Computation

Program	All Unifications				Dynamic Unifications				Dynamic/Assume		
	Unif	%Ind	%S	%F	Dyn	%Ind	%S	%F	Ass	U/Ass	AU/Ass
Extraction	878	89	11	0	165	82	17	1	54	3.05	0.54
Mini-ML	2415	73	11	16	107	87	13	0	10	10.70	1.40
CPS	162	59	41	0	72	57	43	0	48	1.50	0.65
Prop	4957	67	25	8	509	71	14	15	71	7.17	2.10
F-O	1140	69	27	4	27	0	100	0	13	2.08	2.08
Lam	369	50	44	6	36	75	22	3	12	3.00	0.75
DeBruijn	627	20	44	36	77	51	30	19	24	3.21	1.58
CLS	333	30	42	28	0	-	-	-	0	-	-

Program	Disagreement Pairs					Substitutions			
	Tot	%E-?	%G-?	%R	%A	Tot	%Uv	Abs	Abs/Uv
Extraction	1580	22	9	66	6	9016	96	1124	0.01
Mini-ML	5872	17	1	76	6	3644	96	55	0.02
CPS	592	24	34	54	0	1509	100	1029	0.68
Prop	13809	35	3	63	1	12040	99	443	0.04
F-O	6800	21	1	74	5	12716	99	38	0.00
Lam	3464	22	2	74	3	1825	94	83	0.05
DeBruijn	13441	15	1	71	13	14632	99	150	0.01
CLS	5227	23	0	77	0	2	-	0	-

Figure 2: Proof Manipulation

Program	All Unifications				Dynamic Unifications				Dynamic/Assume		
	Unif	%Ind	%S	%F	Dyn	%Ind	%S	%F	Ass	U/Ass	AU/Ass
Comp	5562	90	10	0	1532	92	8	0	67	22.87	1.61
ExpComp	7200	70	10	20	1798	80	8	12	87	10.67	4.30
ExpIndComp	7200	88	10	2	1798	92	8	0	87	10.67	4.30
Trans	2159	70	11	19	107	86	14	0	10	10.70	1.40
ExpTrans	5255	29	10	59	633	15	13	72	67	9.45	8.06
ExpIndTrans	5255	76	11	13	633	84	13	3	67	9.45	8.06

Program	Disagreement Pairs					Substitutions			
	Tot	%E-?	%G-?	%R	%A	Tot	%Uv	Abs	Abs/Uv
Comp	2424	43	0	57	0	445	97	0	-
ExpComp	10765	24	4	56	17	22743	100	778	0.03
ExpIndComp	4251	32	10	52	8	15801	100	778	0.05
Trans	5709	17	1	76	7	3612	96	55	0.02
ExpTrans	27342	20	4	61	16	280679	97	2522	0.01
ExpIndTrans	13482	17	8	65	12	264399	98	2522	0.01

Program	Computation	Transformation
Implicit	1.30	2.48
Explicit	8.48	155.09
Explicit-Indexed	5.80	145.89

Figure 3: Mini-ML comparison

This example involves a large number of level 2 judgments. Indexing is particularly effective here, and assumed rules are used unusually infrequently. Note that these examples do not include any basic computation.

- Mini-ML [13]

  An implementation of Mini-ML, including type-checking, evaluation, and the type soundness proof. Because of the large number of cases, indexing has a stronger effect than in all other examples.

- CPS — *Interpretation of propositional logics and CPS conversions* [3, 23]

  Various forms of conversion of simply-typed terms to continuation-passing and exception-returning style. Substitutions are all parameter substitutions, and unification involves an unusually large number of gvar-anything cases. The redundant type computations are very significant in this example—all the examples are level 2 judgments.

- Canonical — *Canonical forms in the simply-typed lambda-calculus* [21]

  Conversion of lambda-terms to canonical form. A small number of non-parameter substitutions arise, but mostly unification is first-order. Here, too, there is much redundant type computation.

- Prop — *Propositional Theorem Proving and Transformation* [5]

  This is mostly first-order. In the transformations between various proof formats (natural deduction and Hilbert calculi), a fairly large number of assumptions arise, and are quite heavily used. Unification involves a large number of evar-anything cases.

- F-O — *First-order logic theorem proving and transformation*

  This includes a logic programming style theorem prover and transformation of execution trace to natural deductions. There is rather little abstraction.

- Forsythe — *Forsythe type checking*

  Forsythe is an Algol-like language with intersection types developed by Reynolds [24]. This example involves very few substitutions, all of which are parameter substitutions. Thus the runtime behavior suggests an almost entirely first-order program, which is not apparent from the code.

- Lam — *Lambda calculus convertibility*

  Normalization and equivalence proofs of terms in a typed λ-calculus. A relatively high percentage of the substitutions are non-parameter substitutions.

- Polylam — *Type inference in the polymorphic lambda calculus* [20]

  Type inference for the polymorphic λ-calculus involves postponed constraints, but mostly parameter substitutions. Unification can be highly non-deterministic. This is not directly reflected in the given tables, as this is the only one of our examples where any hard constraints

are delayed at run time (and in only 10 instances). In fact, one of these hard constraints remains all the way to the end of the computation. This indicates that the input was not annotated with enough type information (within the polymorphic type discipline, not within the framework).

- Records — *A lambda-calculus with records and polymorphism*

  Type checking for a lambda-calculus with records and polymorphism as described in [6]. This involves only parameter substitutions, and assumptions are heavily used.

- *DeBruijn* [4]

  A compiler from untyped λ-terms to terms using deBruijn indices, including a call-by-value operational semantics for source and target language. The proof manipulation queries check compiler correctness for concrete programs. Indexing works quite poorly, and an unusually large number of abst-abst cases arise in unification.

- *CLS* [4]

  A second compiler from terms in deBruijn representation to the CLS abstract machine. Simple queries execute the CLS machine on given programs, proof manipulation queries check compiler correctness for concrete programs. This is almost completely first-order.

Overall, the figures suggest quite strongly that most unification is either simple assignment or first-order (Herbrand) unification, around 95%, averaged over all examples. Similarly, substitution is the substitution of parameters for λ-bound variables in about 95% of the cases. The remaining 5% are substitution of constants, variables, or compound terms for bound variables. These figures do not count the substitution that may occur when clauses are copied, or unifications or substitutions that arise during type reconstruction.

Finally, we compare the Mini-ML program with a version written using explicit substitution, to evaluate the effects of a syntactic restriction along the lines of L_λ. The computation queries had to be cut down somewhat because of memory restrictions. In Figure 3 we show the same data as above for the computation and transformation queries with and without explicit substitution. We also show a version with explicit substitution with the substitution code rewritten to take better advantage of indexing. Then we compare the CPU times (in seconds) for the two sets of queries for all three versions of the program, using a slightly modified[3] Elf version 0.2 in SML/NJ version 0.80 on a DEC station 5000/200 with 64MB of memory and local paging. These results show that there is a clear efficiency disadvantage to the L_λ restriction, given present implementation techniques. Note that the disadvantage is greater for the transformation queries, since a longer proof object is obtained, resulting in a more complicated proof transformation. Explicit substitution increases the size of the relevant code by 30%.[4] Substitutions dominate the computation time, basically because one meta-level β-reduction has been replaced by many substitutions. These substitutions

[3] The modification involves building proof objects only when needed for correctness.

[4] Actually, the meta-theory was not completely reduced to L_λ, because type dependencies in the verification code would lead to a very complex verification predicate. We estimate that the code size would increase an additional 5% and the computation time by much more than that.

should all be parameter (uvar) substitutions, which suggests that some (but clearly not all) of the performance degradation could be recovered through efficient uvar substitution. See the previous footnote on why non-parameter substitutions still arise in the proof transformation examples.

3.2 Further Summary Analysis

A few figures were obtained through simple summary profiling and await further detailed analysis. The summary figures suggest that, for examples of average size, omitting the (extended) occurs-check in the current implementation can result in speed improvements of between 40% and 60%. This is therefore an upper bound on the speed-up that could be achieved through smart compilation to avoid the occurs-check.

The current implementation avoids building proof objects to some extent (applicable to Elf only), which saves about 50% of total computation time, although the savings are not additive (some of the occurs-check overhead arises in building proofs).

4 Conclusions

We briefly summarize our preliminary conclusions, which are very much in line with the experience gained in other constraint logic programming languages [12].

Language Design. Statically prohibiting difficult cases in unification (by a restriction to L_λ, for example) is not a good idea, since it leads to a proliferation of code and significantly complicates meta-theory as it is typically expressed in Elf. This coincides with experience in other constraint logic programming languages such as CLP($\mathcal{R}$) and Prolog-III.

Our recommendation is to delay hard constraints (including flexible-rigid pairs that are not gvar-rigid pairs) and thus avoid branching in unification at runtime.

Language Implementation. Indexing and representation of terms in the functor/arg notation (rather then the curried notation typical for λ-calculi) are crucial for achieving good performance, as they enable quick classification of disagreement pairs and rigid-rigid decomposition. It is rather obvious that runtime type computation must be avoided whenever possible as suggested in [11], and that proof building must be avoided whenever the proof object will not be needed.

We need special efficient mechanisms for direct binding and first-order unification. Furthermore, unification as in L_λ and substitution of parameters for bound variables are very important special cases that merit special attention. Efficiency of substitution of constants or compound terms for bound variables is important in some applications, but not nearly as pervasive and deserves only secondary consideration.

5 Future Work

A study such as this is necessarily restricted and biased by the currently available implementation technology. The most important figures that are currently missing:

- How much type computation can be eliminated, and what would be the effect of eliminating redundant type computation on the remaining figures.

- How often can the occurs-check be avoided.

In longer term work, one would also like to analyse the effect of other standard compilation techniques of logic programming languages in this new setting, but much of this requires an implemented compiler as a basis.

References

[1] Penny Anderson. *Program Development by Proof Transformation.* PhD thesis, Carnegie Mellon University, 1992. In preparation.

[2] Scott Dietzen and Frank Pfenning. Higher-order and modal logic as a framework for explanation-based generalization. *Machine Learning*, 9:23–55, 1992.

[3] Timothy G. Griffin. Logical interpretations as computational simulations. Draft paper. Talk given at the North American Jumelage, AT&T Bell Laboratories, Murray Hill, New Jersey, October 1991.

[4] John Hannan and Frank Pfenning. Compiler verification in LF. In Andre Scedrov, editor, *Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 407–418, Santa Cruz, California, June 1992. IEEE Computer Society Press.

[5] Robert Harper and Frank Pfenning. A module system for a programming language based on the LF logical framework. Technical Report CMU-CS-92-191, Carnegie Mellon University, Pittsburgh, Pennsylvania, September 1992.

[6] Robert Harper and Benjamin Pierce. A record calculus based on symmetric concatenation. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 131–142, Orlando, Florida, January 1991.

[7] Joshua S. Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 199? To appear. A preliminary version appeared in the Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science, pages 32–42, Amsterdam, The Netherlands, July 1991.

[8] Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[9] Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. The CLP($\mathcal{R}$) language and system. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, July 1992.

[10] Joxan Jaffar, Spiro Michaylov, and Roland Yap. A methodology for managing hard constraints in CLP systems. In Barbara Ryder, editor, *Proceedings of the ACM SIGPLAN Symposium on Programming Language Design and Implementation*, pages 306–316, Toronto, Canada, June 1991.

[11] Keehang Kwon, Gopalan Nadathur, and Debra Sue Wilson. Implementing logic programming languages with polymorphic typing. Technical Report CS–1991–39, Duke University, Durham, North Carolina, October 1991.

[12] Spiro Michaylov. *Design and Implementation of Practical Constraint Logic Programming Systems.* PhD thesis, Carnegie Mellon University, August 1992. Available as Technical Report CMU-CS-92-168.

[13] Spiro Michaylov and Frank Pfenning. Natural semantics and some of its meta-theory in Elf. In L.-H. Eriksson, L. Hallnäs, and P. Schroeder-Heister, editors, *Proceedings of the Second International Workshop on Extensions of Logic Programming*, pages 299–344, Stockholm, Sweden, January 1991. Springer-Verlag LNAI 596.

[14] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. In Peter Schroeder-Heister, editor, *Extensions of Logic Programming: International Workshop, Tübingen FRG, December 1989*, pages 253–281. Springer-Verlag LNCS 475, 1991.

[15] Dale Miller. Unification of simply typed lambda-terms as logic programming. In K. Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming*, pages 255–269. MIT Press, July 1991.

[16] Gopalan Nadathur and Bharat Jayaraman. Towards a WAM model for lambda Prolog. In *Proceedings of the 1989 North American Conference on Logic Programming*, pages 1180–1198. MIT Press, October 1989.

[17] Gopalan Nadathur and Dale Miller. An overview of λProlog. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium, Volume 1*, pages 810–827, Cambridge, Massachusetts, August 1988. MIT Press.

[18] Gopalan Nadathur and Debra Sue Wilson. A representation of lambda terms suitable for operations on their intensions. In *Proceedings of the 1990 Conference on Lisp and Functional Programming*, pages 341–348. ACM Press, June 1990.

[19] Pascal Brisset Olivier Ridoux, Serge Le Huitouze. Prolog/mali. Available via ftp over the Internet, March 1992. Send mail to pm@irisa.fr for further information.

[20] Frank Pfenning. Partial polymorphic type inference and higher-order unification. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 153–163, Snowbird, Utah, July 1988. ACM Press.

[21] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[22] Frank Pfenning. Unification and anti-unification in the Calculus of Constructions. In *Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 74–85, Amsterdam, The Netherlands, July 1991.

[23] Frank Pfenning and Ekkehard Rohwedder. Implementing the meta-theory of deductive systems. In D. Kapur, editor, *Proceedings of the 11th International Conference on Automated Deduction*, pages 537–551, Saratoga Springs, New York, June 1992. Springer-Verlag LNAI 607.

[24] John C. Reynolds. Preliminary design of the programming language Forsythe. Technical Report CMU-CS-88-159, Carnegie Mellon University, Pittsburgh, Pennsylvania, June 1988.

A Proposal for Modules in λProlog: Preliminary Draft

Dale Miller [1]
Computer Science Department
University of Pennsylvania
Philadelphia, PA 19104-6389 USA
dale@saul.cis.upenn.edu

Abstract

Higher-order hereditary Harrop formulas, the underlying logical foundation of λProlog [20], are more expressive than first-order Horn clauses, the logical foundation of Prolog. In particular, various forms of scoping and abstraction are supported by the logic of higher-order hereditary Harrop formulas while they are not supported by first-order Horn clauses. Various papers have argued that the scoping and abstraction available in this richer logic can be used to provide for modular programming [15], abstract data types [14], and state encapsulation [7]. None of these papers, however, have dealt with the problems of *programming-in-the-large*, that is, the essentially linguistic problems of putting together various different textual sources of code found, say, in different files on a persistent store into one logic program. In this paper, I propose a module system for λProlog and shall focus mostly on it static semantics. The dynamic aspects are covered in various other papers: in particular, see the paper by Kwon, Nadathur, and Wilson [10] in these proceedings.

1 Module syntax should be declarative

Several modern programming languages are built on declarative, formal languages: for example, ML and Scheme are based on the λ-calculus and Prolog is based on Horn clauses. Initial work on developing such languages was first concerned with programming-in-the-small: problems with programming-in-the-large were attached later. At that point, a second language was often added on top of the initial language. For example, parsing and compiler directives, such as `use`, `import`, `include`, and `local`, were added. This second language generally had little connection with the original declarative foundation of the initial language: its was born out of the necessity to build large programs and its function was expediency. The meaning of the resulting hybrid language is often complex since it loses some of its declarative purity.

Occasionally, programming design is inflicted with what we may call the "recreating the Turing machine" syndrome. Turing machines were important because they were the first formal system that obviously computed and were clearly easy to implement. They have not been considered seriously as programming languages for several reasons, including the difficulty of understanding and

[1] Supported in part by ONR N00014-88-K-0633, NSF CCR-91-02753, and DARPA N00014-85-K-0018.

reasoning about transition tables. Often the development of modular constructions in programming languages follows a similar path: it is generally easy to develop a language for programming-in-the-large that obviously separates and hides details and for which efficient implementations are possible. Often, however, it is difficult to reason about the meaning of the resulting language.

In order to avoid this syndrome we should ask that any proposal for programming-in-the-large have several high-level principles. For example, we should ask for such proposals to support several of the following properties.

- There should be a non-trivial notion of the equivalence of modules that would guarantee that a module can be replaced by an equivalent module with little to no impact on the behavior of a larger program. This property is sometimes called *representation independence* (see Section 3).
- Constructs for programming-in-the-large should not complicate the meaning of the underlying, declarative language.
- Modules should support transitions from specification to implementation.
- Modular programming should work smoothly with higher-order programming. In Prolog, a particular challenge is getting the semantics of the `call/1` predicate correct.
- Rich forms of abstraction, hiding, and parametrization should be possible.
- Modules should allow a rich calculus of transformations. These should include partial evaluation, fold/unfold, and even compilation.
- Important aspects of a module's meaning should be available and verified without examining the module in detail. Notions of interfaces often support this property.
- The additional syntax for programming-in-the-large should also be readable, natural, and support separate compilation and re-usability.

The success of a proposal for modular programming should not be judged simply on its obviousness or easy of implementation: it should also be judged on its ability to support a large number of properties such as these.

One approach: map module syntax directly to logic There are some logical systems that can be used as a basis of logic programming and that contain natural notions of scope for program clauses and constants. For example, the logic of *hereditary Harrop formulas*, parts of which were developed independently by Gabbay and Reyle [4], McCarty [11, 12], and Miller [13, 15, 16], allows for a simple stack-based structuring of the runtime program and set of constants. The modal logic of Giordano, Martelli, and Rossi [5] provides an interesting variation on the simple "visibility rules" effecting logic programs based on the intuitionistic theory of hereditary Harrop formulas. A recent linear logic refinement of hereditary Harrop formulas by Hodas and Miller [8] modifies the stacked-based discipline of programs by allowing some program clauses to be deleted once they are used within a proof.

One approach to developing a principled modular programming language is to reduce programming-in-the-large to programming-in-the-small is such a way that modular programming can be explained completely in terms on the logical connectives of the underlying language. That is, a linked collection of modules would be mapped to a (possibly large) collection of (possibly large) formulas. Furthermore, we would like the combinators for building modules to correspond closely to logical connectives. The *static semantics* of a collection of modules is specified by describing how such modules denote a collection of constants and program clauses. The *dynamic semantics* of a collection of modules is specified by describing the collection of goal formulas that can be proved from them. Given the richness of hereditary Harrop formulas and their variants, the main challenge in specifying the static semantics of modules appears to be determining the scope and types of constants.

2 A specific module proposal

We shall now turn to a specific proposal for modules for λProlog. Since the underlying logic of λProlog is that of the intuitionistic (actually minimal) theory of hereditary Harrop formulas, we shall consider how modules can be mapped into such formulas. It would be interesting to consider a similar mapping into either the modal or linear logic variants of these formulas mentioned above. We shall not, however, consider these other variations here.

2.1 General comments

λProlog extends first-order Horn clauses in several ways. As it turns out, much of the scoping primitives for the module facility proposed here do not come from the higher-order quantification available in λProlog. In fact, the propositional logic fragment of λProlog supports the stacked-based treatment of programming clauses. Higher-order quantification is important, however, in providing scope for predicate and function symbols as well as in providing for higher-order programming (an important abstraction separate from the module proposal here.

Both the proof theoretic and model theoretic treatments of λProlog's foundation treats a program as a pair containing a signature and set of clauses. For example, the proof theoretic treatment of λProlog given in [16] uses sequents of the form $\Sigma; \mathcal{P} \longrightarrow G$, where Σ is a signature (a collection of typed constants) and $\mathcal{P}$ is a set of Σ-formulas (closed formulas all of whose non-logical constants are contained in Σ). Similarly, a canonical model for a large fragment of the logic underlying λProlog can be given as a Kripke model where possible worlds are pairs $\langle \Sigma, \mathcal{P} \rangle$, where Σ is a signature and $\mathcal{P}$ is a set of Σ-formulas [17]. Thus it will not be surprising that the module proposal presented here will make extensive use of signatures. Even if λProlog was not a typed language signatures would be important since the set of constants available to a computation changes, and describing how that set of constants change would make use of a notion of signature similar to that used here. Gunter [6] also makes use of signatures in developing a module calculus for λProlog.

Finally, it is important to say that what follows is just the draft of a proposal. Much of what follows has not be debated by those currently using implementations of λProlog. Also, most experience with λProlog has been with small programs. Few people have yet had experience with

large λProlog programs. This proposal is hopefully another step in determining a viable solution to programming-in-the-large in this logic programming setting.

2.2 Persistent store

Interacting with a persistent store, such as the Unix file system, is problematic within our logic programming setting: some non-logical predicates are required at the core of our module facility. In particular, the predicate

```
type load  string -> o
```

predicate performs a side-effect: it is used to reflect some of the persistent store into the space of meaningful λProlog objects. As edits are done on files, new calls to `load` are needed to update these objects. An attempt to prove the atom `load name` takes the string `name` as a reference to an actual file. The resolution of this string into a file can be done in possibly many ways. The method used in LP2.7 [18] was to maintain a list of Unix path names and to search in them for a file whose name is `name` augmented with ".mod". If such a file is found, then it is parsed and type checked. Other methods to resolve the string `name` with a file are possible.

2.3 Kinds and types

In order to allow useful types, we admit type constructors. There is only one of these built into λProlog, namely the infix "function space" constructor `->`. Other type constructors can be declared via the `KIND` declaration. (Keywords will be capitalized for readability: in most implementations of λProlog, keywords appear in lowercase letters.) For example,

```
KIND bool type.
KIND list type -> type.
KIND pair type -> type -> type.
```

As this example show, the only kind that can be associated with a type constructor is any "first-order kind" involving only `type` and `->`. Qualifying a type constructor with a non-negative integer (0 instead of `type`, 1 instead of `type -> type`, etc.) could also have worked here.

Types will be used to qualify constants. Types are any first-order term structure built from type variables and type constructors. The presence of types variables will provide λProlog with a degree of polymorphism. Type variables are tokens within type expressions that have an initial uppercase letter. The following are some type declarations.

```
TYPE nil    list A.
TYPE ::     A -> list A -> list A.
TYPE append list A -> list A -> list A -> o.
TYPE memb   A -> list A -> o.
```

λProlog has numerous build-in types, including type `o`, the type of λProlog formulas.

The subsumption relation on types is that familiar from first order logic: a type is subsumed by another type if the first is a substitution instance of the second.

2.4 Static semantics for types and terms

I will assume that types are property formed (they respect kind declarations) and that formulas and terms are well typed. See [21] for a fuller discussion of this aspect of static semantics.

2.5 Signatures

Signatures are lists of tokens assigned kinds and types, and are denoted by the syntactic variable Σ. The same token can be given a type and a kind. Op-declarations are also stored as members of signatures. The following is an example of a signature.

```
OP   150 :: xfy.
KIND list           type -> type.
TYPE ::             A -> list A -> list A.
TYPE nil            list A.
TYPE memb, member   A -> list A -> o.
TYPE append, join   list A -> list A -> list A -> o.
```

A formula is a Σ-formula if it is a correctly typed, closed formula all of whose non-logical constants are from Σ. Since modules are collections of formulas, we shall use signatures to qualify (type) modules.

It will be useful to have *signature descriptions* to represent possibly long lists of constants. For this, we shall use the keywords SIGNATURE, TYPE, KIND, OP, ACCUMULATE, LOCAL, and LOCALKIND. The keyword SIGNATURE is used to name a signature and the keywords TYPE, KIND, and OP are used simply to enumerate the members of a signature. ACCUMULATE takes a list of signatures: its intended meaning is to merge in the listed signatures. The two keywords LOCAL and LOCALKIND are used to limit the scope of types and kinds so that they are actually not part of this signature. The LOCAL keyword can take a type declaration as an optional third argument; similarly with LOCALKIND. The following are two signature descriptions.

```
SIGNATURE lists.
OP   150 :: xfy.
KIND list           type -> type.
TYPE ::             A -> list A -> list A.
TYPE nil            list A.
TYPE memb,member    A -> list A -> o.
TYPE append, join   list A -> list A -> list A -> o.

SIGNATURE rev.
ACCUMULATE lists.
TYPE reverse list A -> list A -> o.
LOCAL revaux list A -> list A -> list A -> o.
LOCAL join.
```

Constants can be given multiple types within the same module or within ACCUMULATEing chains of modules. It is an error if these types are not comparable via subsumption. Otherwise, the type assumed is the least general of those types.

Signature descriptions are *elaborated* into signatures using the following rules. First, eliminate all ACCUMULATE keywords by replacing them with the signatures they name. In doing this, if a constant is given two op-declarations, then it is an error if those two declarations are not identical. Second, LOCAL can be dropped by deleting it and any constant of the same name in the accumulated signature. If LOCALKIND is present, then first check to see if there are constants in the signature that have a type containing this type constructor. If so, produce an error. Otherwise, simply drop this declaration.

The notion of *signature containment* is given simply as follows: Σ_1 is contained in Σ_2 if

- for every constant in Σ_1 given a kind, that constant is given the same kind in Σ_2,
- for every constant in Σ_1 given a type τ, that constant is given a type in Σ_2 that subsumes τ, and
- for every constant in Σ_1 given an op-declaration, that constant is given the identical op-declaration in Σ_2.

This notion of signature containment will be needed for defining equal signatures and for a certain kind of dynamic qualification of modules (see subsection 2.10).

We shall assume that there is a special system signature that contains declarations for all logical and built-in constants of a given λProlog system.

2.6 Module syntax

Modules will be built from kinds, types, and program clauses using the following keywords: TYPE, KIND, OP, LOCAL, LOCALKIND, MODULE, ACCUMULATE, and IMPORT. The meaning of TYPE, KIND, and OP are as they were for signature descriptions. The keyword MODULE names a module (similar to the keyword SIGNATURE). The keywords LOCAL and LOCALKIND provide scope to constants within a module: the dynamic semantics of LOCAL will be interpreted as an existential quantifier, as described in [14]. The keywords ACCUMULATE and IMPORT will be described further below.

Although only the keyword MODULE must appear at the front of a module, for the convenience of parsing and reading modules, we assume that it is an error if a declaration of a constant appears after the first occurrence of that constant. All declarations are global in a module. Figure 1 contains two examples of modules.

2.7 Static semantics for modules

The static semantics of modules is used to determine which signature and formulas are intended by the module. Since we are attempting to reduce modules to formulas, recursion between modules is not allowed: that is, if mod1 imports or accumulates mod2 then mod2 can not import or accumulate mod1.

A signature description is built from a module as follows.

```
MODULE lists.

OP 150 :: xfy.
KIND list         type -> type.

TYPE ::           A -> list A -> list A.
TYPE nil          list A.
TYPE memb,member  A -> list A -> o.
TYPE append, join list A -> list A -> list A -> o.

memb X (X::L).
memb X (Y::L) :- memb X L.

member X (X::L) :- !.
member X (Y::L) :- member X L.

append nil K K.
append (X::L) K (X::M) :- append L K M.

join nil K K.
join (X::L) K M :- memb X K, !, join L K M.
join (X::L) K (X::M) :- join L K M.
```

```
MODULE rev.

ACCUMULATE lists.
TYPE  reverse  list A -> list A -> o.
LOCAL rev      list A -> list A -> list A -> o.

reverse L K :- rev L K nil.

rev nil K K.
rev (X::L) K (X::Acc) :- rev L K ACC.
```

Figure 1: The `lists` and `rev` modules.

- TYPE and KIND declarations stay TYPE and KIND declarations.
- All IMPORTed, ACCUMULATEd, and module implication (==>) modules have their signatures ACCUMULATEd.
- If the qualified module importing (===> mod sig) G is used, then the signature **sig** is ACCUMULATEd (see Section 2.10 for a description of ===>).
- LOCAL and LOCALKIND become LOCAL and LOCALKIND.

Notice that it is possible for LOCAL and LOCALKIND to provide scope to a constant that is IMPORTed or ACCUMULATEd. If IMPORT or ACCUMULATE is used in a module and there is no corresponding module with the correct name, then look for a signature with that name. Thus modules without clauses can simply be written as signatures.

The static semantics of the IMPORT keyword construction is a bit complicated, although it does follow closely the lines described in [15] and implemented in LP2.7 and eLP [3]. If a module mod1 contains the line

```
IMPORT mod2 mod3.
```

then the modules mod2 and mod3 are made available (via implications) during the search for proofs of the body of clauses listed in mod1. Thus, if the formulas E_2 and E_3 are associated with mod2 and mod3, then a clause $G \supset A$ listed in mod1 is elaborated to the clause $((E_2 \wedge E_3) \supset G) \supset A$.

Notice that a module denotes both a set of program clauses and a signature. The signature that is inferred from a module can be used as an interface: when parsing and compiling modules, it should only be necessary for the signature of an accumulated or imported module to be read.

2.8 Environment support

The process of parsing a module will also be accompanied with type checking and type inference. In particular, a file containing a module may not attribute a type to all constants. In this case, the programming environment must be able to infer a reasonable type for the undeclared constants. Type inference can be done much as it is in ML: see [21] for more discussion on type inference for λProlog.

Signature checking and inference will also need to be done by the environment. Checking involves making certain that when modules are accumulated and imported, constants are not given incomparable types and declarations.

2.9 Dynamic semantics for modules

I shall assume that the reader is already familiar with the operational (dynamic) semantics of hereditary Harrop formulas, in particular, with the meaning of implications and universal quantifiers in goals.

The ACCUMULATE keyword. Although the meaning of this keyword is simple, it is not present in either LP2.7 or eLP. It is similar to the `use` directive of Prolog/Mali. If a module `mod1` contains the line

```
ACCUMULATE mod2 mod3.
```

then is intended that the program clauses in `mod2` and `mod3` are available at the end of the list of program clauses listed explicitly in `mod1`.

The IMPORT keyword. Proof search based on clauses obtained by importing a module into another module can benefit from some recent work on provability in intuitionistic logic. For example, both Hudelmaier [9] and Dyckhoff [2] have demonstrated that the implication-left rule can be improved (with respect to proof search). For example, the implication-left rule can be split into several cases depending of the form of the implication. The following is one of these rules.

$$\frac{\Sigma; \mathcal{P}, E, G \supset D \longrightarrow G \qquad \Sigma; \mathcal{P}, D \longrightarrow G'}{\Sigma; \mathcal{P}, (E \supset G) \supset D \longrightarrow G'}$$

Consider the case when the formulas D and G' are the same atomic formula A.

$$\frac{\Sigma; \mathcal{P}, E, G \supset A \longrightarrow G}{\Sigma; \mathcal{P}, (E \supset G) \supset A \longrightarrow A}$$

Notice that the formula $(E \supset G) \supset A$ could be the result of importing a module E into a module listing the clause $G \supset A$. Notice that backchaining on a clause in this module provides an operational reading of importing: the imported module is added to the current clauses along with the un-elaborated clauses from the initial module.

A generalization of this inference rule would be the following:

$$\frac{\Sigma; \mathcal{P}, E, \wedge_{i=1}^{n}(G_i \supset A_i) \longrightarrow G_j}{\Sigma; \mathcal{P}, \wedge_{i=1}^{n}((E \supset G_i) \supset A_i) \longrightarrow A}$$

where A_j is equal to A, for some $j = 1, \ldots, n$. An argument for the completeness for this rule can be found in [10].

In the above inference rule, assume that the formula E is of the form $\exists \bar{x}.D$ where the list of typed, bound variables $\bar{x}$ are not in the signature Σ. This inference rule could then be modified to be

$$\frac{\Sigma, \bar{x}; \mathcal{P}, D, \wedge_{i=1}^{n}(G_i \supset A_i) \longrightarrow G_j}{\Sigma; \mathcal{P}, \wedge_{i=1}^{n}((E \supset G_i) \supset A_i) \longrightarrow A}.$$

Thus, backchaining into a module which imports a module containing local constants essentially loads its local constants into the current signature and loads it's code (the formula D) into the current program.

Another important aspect of the dynamic semantics of modules is presented in [10] where the AUGMENT search rule is modified to be the AUGMENT' search rule. This new rule is used only for modules and not formulas thus forcing an operational (but not declarative) distinction between

programming-in-the-large and small. The AUGMENT' rule essentially says that if the current program space already contains a module, that module should not be assumed again: that is, there should be at most one copy of a module in the current program space at a time. The goal `mod ==> mod ==> G` is operationally the same as `mod ==> G`. Such an optimization is unlikely at the level of formulas because of the following example. Consider a goal of the form `(p a) => (p X) => G`, where `X` is a logical variable. If we checked to see if `(p X)` in the context, it would seem that we should allow the unification of `X` with `a`. It would be easy to construct examples where the order of instantiating variables would yield two different answers to this computation, an undesirable effect.

2.10 Questions and additional features

I list below some questions and possible additional features that could be incorporated in the module system sketched above.

Parametric modules When a module is defined using the `MODULE` keyword, it might be possible to also add to it a signature over which that module is parametric. An example could be given as follows.

```
MODULE {quicksort KIND Atype    type.
                  TYPE Order    Atype -> Atype -> o}.

TYPE      qsort  list Atype -> list Atype -> o.
LOCAL     split  Atype -> list Atype -> list Atype -> list Atype -> o.
IMPORT    lists.

qsort nil nil.
qsort (X::L) K :- split X L Low High, qsort Low R,
                  qsort High S, append R (X::S) K.

split X (Y::L) (Y::K) M :- Order X Y, !, split X L K M.
split X (Y::L) K (Y::M) :- split X L K M.
```

The argument signature is described using only the `KIND` and `TYPE` keywords and the order in which items are listed in this signature is important. The corresponding signature should probably be written as

```
SIGNATURE {quicksort KIND Atype    type.
                     TYPE Order    Atype -> Atype -> o}.

TYPE        qsort  list Atype -> list Atype -> o.
ACCUMULATE  lists.
```

A use of such a module can be given as

```
?- {quicksort int <} ==> qsort (2::3::4::nil) L.
```

Parsing this "module implication" `==>` is a bit different from parsing other terms, in particular, the subexpression `{quicksort int <}` should be treated by the parser as a subterm over the signature

```
KIND int   type.
TYPE >     int -> int -> o.
TYPE qsort int list -> int list -> o.
```

plus the signature items in `lists` (and the system module, where `<` is given an op-declaration).

Using constants to denote modules and signatures. The names for modules and signatures should be converted to constants that are given types, say `modname` and `signame`, and declarations for these names need to be added (destructively) to the system module. In this way, they will be available globally. Thus, `==>` and `===>` (this second arrow is described below) would have the types

```
KIND modname, signame    type.
TYPE ==>  modname -> o -> o.
TYPE ===> modname -> signame -> o -> o.
```

The current convention in LP2.7 and eLP is that there is one module per file and that the file's name is built from the module's name. This approach has the advantage that by mentioning a module name in one of these interpreters, it is possible for the system to find the file containing that module. It may be an advantage, however, to drop this linkage, in which case, files, possibly containing a number of modules and signatures, are loaded by using entire path names. For the purposes of compilation and parsing, once a file is parsed and checked, a second, parallel file containing only signatures might be generated from the one that is just parsed. It should only be this second file that is needed during parsing and compiling of other modules. The `aux` files generated by Prolog/Mali [1] are essentially signatures that parallel modules.

Quantification over module names. It may be possible to permit variables to range over modules if we are willing to admit runtime signature checking of modules. For example, consider a goal of the form `(===> mod sig G)`. Here `mod` is a module whose signature is contained in that given by `sig`: this check would be done when this goal is attempted. Thus, in determining the static properties of a goal with this syntax, simply use the signature `sig` instead of attempting to determine the one for `mod`, which may be a variable. Thus, a goal of the form

```
?- memb M (mod1::mod2::mod3::nil), ===> M sig G.
```

would search for a module that can be used to establish the goal `G`. If all the modules `mod1`, `mod2`, and `mod3` have a signature contained in the signature `sig`, then no runtime error is generated by this goal. The syntax `(===> M sig G)` is essentially the same as `(M ==> G)` except that `M` must be restricted by the signature `sig`. Notice that it will not be possible to quantify over signature names.

Other declarations. Other declarations besides those for op might also be allowed. For example, certain types could be specified as being open or closed and certain predicates could have declarations describing how atomic goals could be suspended if certain argument positions are unbound.

Relationship to other aspects of an interpreter. The interaction of the module system with input/output and with the top-level of an interpreter must also be considered carefully.

3 Formal aspects of this proposal

The design of λProlog has been motivated in part by the desire to make logic play as large a role as possible in efforts to extend the expressiveness of logic programming. There are many reasons for this emphasis on logic: the resulting language remains declarative and programs can be given meaning using such deep meta-theoretic properties as cut-elimination and model theoretic semantics. Thus, analyzing *programming-in-the-small* within "pure" λProlog can be attached using these deep principles. We can hope that the language for describing modules will also have such principles.

As an example of such principles, consider the problem of *representation independence* for abstract data types. If we follow the line of argument given in [14] (and above) for coding abstract data types, representation independence follows directly. For example, consider the following two existentially quantified formulas, E_1 and E_2, which provide different implementations of queues. (I shall use the syntactic variable E to range over possibly existentially quantified definite formulas.)

```
sigma qu\(sigma f\(
     pi L\        ( empty (qu L L) ),
  pi X\(pi L\(pi K\( enter  X (qu L (f X K)) (qu L K) ))),
  pi X\(pi L\(pi K\( remove X (qu (f X L) K) (qu L K) ))) )).

sigma emp\(sigma g\(
                  ( empty emp ),
  pi X\(pi L\     ( enter  X L (g X L) )),
  pi X\           ( remove X (g X emp) emp ),
  pi X\(pi L\(pi K\( remove X (g Y L) (g Y K) :- remove X L K ))) )).
```

Let $\vdash$ be intuitionistic provability and let $\vdash^+$ be an enrichment of $\vdash$ that is conservative over $\vdash$ and that also makes it possible to reason about data structures (that is, induction must be incorporated). Then if we show that E_1 and E_2 are equivalent in $\vdash^+$, that is, $E_1 \vdash^+ E_2$ and $E_2 \vdash^+ E_1$, then the following argument is immediate: if $\Gamma, E_1 \vdash G$ then $\Gamma, E_1 \vdash^+ G$ since $\vdash^+$ enriches $\vdash$; by cut-elimination (assumed also for $\vdash^+$), $\Gamma, E_2 \vdash^+ G$; finally, by conservative extension, $\Gamma, E_2 \vdash G$. Thus, if a goal G is provable using E_1, it is provable using E_2 (the converse is similar). The fact that abstractions are based on logic made this argument particularly direct.

Since the higher-order theory of hereditary Harrop formulas has been worked out in [19], there should be little problem getting this module facility to work smoothly with higher-order programming. Numerous other formal aspects of this module proposal must also be explored.

4 Conclusion

I have described a possible approach to programming-in-the-large for λProlog. This proposal is designed to ensure that the module constructions are declarative and this was done by making certain that the module syntax can be replaced in a very natural way by logical connectives.

This proposal is just a draft: many details have been left out. A subsequent version of this proposal will hopefully correct this shortcoming.

References

[1] Pascal Brisset and Olivier Ridoux. The architecture of an implementation of λProlog: Prolog/Mali. In *Proceedings of the 1992 λProlog Workshop*, 1992.

[2] Roy Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *Journal of Symbolic Logic*, 57(3), September 1992.

[3] Conal Elliott and Frank Pfenning. eLP, a Common Lisp Implementation of λProlog. Implemented as part of the CMU ERGO project, May 1989.

[4] D. M. Gabbay and U. Reyle. N-Prolog: An extension of Prolog with hypothetical implications. I. *Journal of Logic Programming*, 1:319 – 355, 1984.

[5] L. Giordano, A. Martelli, and G. F. Rossi. Local definitions with static scope rules in logic languages. In *Proceedings of the FGCS International Conference, Tokyo*, 1988.

[6] Elsa L. Gunter. Extensions to logic programming motivated by the construction of a generic theorem prover. In Peter Schroeder-Heister, editor, *Extensions of Logic Programming: International Workshop, Tübingen FRG, December 1989*, volume 475 of *Lecture Notes in Artificial Intelligence*, pages 223–244. Springer-Verlag, 1991.

[7] Joshua Hodas and Dale Miller. Representing objects in a logic programming language with scoping constructs. In David H. D. Warren and Peter Szeredi, editors, *1990 International Conference in Logic Programming*, pages 511 – 526. MIT Press, June 1990.

[8] Joshua Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Journal of Information and Computation*, 1992. Invited to a special issue of papers from the 1991 LICS conference.

[9] Jörg Hudelmaier. *Bounds for cut elimination in intuitionistic propositional logic.* PhD thesis, University of Tübingen, Tübingen, 1989.

[10] Keehang Kwon, Gopalan Nadathur, and Debra Sue Wilson. Implementing a notion of modules in the logic programming langauage λprolog. In *Proceedings of the 1992 λProlog Workshop*, 1992.

[11] L. T. McCarty. Clausal intuitionistic logic I. fixed point semantics. *Journal of Logic Programming*, 5:1 – 31, 1988.

[12] L. T. McCarty. Clausal intuitionistic logic II. tableau proof procedure. *Journal of Logic Programming*, 5:93 – 132, 1988.

[13] Dale Miller. A theory of modules for logic programming. In Robert M. Keller, editor, *Third Annual IEEE Symposium on Logic Programming*, pages 106 – 114, Salt Lake City, Utah, September 1986.

[14] Dale Miller. Lexical scoping as universal quantification. In *Sixth International Logic Programming Conference*, pages 268–283, Lisbon, Portugal, June 1989. MIT Press.

[15] Dale Miller. A logical analysis of modules in logic programming. *Journal of Logic Programming*, 6:79 – 108, 1989.

[16] Dale Miller. Abstractions in logic programming. In Peirgiorgio Odifreddi, editor, *Logic and Computer Science*, pages 329 – 359. Academic Press, 1990.

[17] Dale Miller. Abstract syntax and logic programming. In *Logic Programming: Proceedings of the First and Second Russian Conferences on Logic Programming*, number 592 in Lecture Notes in Artificial Intelligence, pages 322–337. Springer-Verlag, 1992. Also available as technical report MS-CIS-91-72, UPenn.

[18] Dale Miller and Gopalan Nadathur. λProlog Version 2.7. Distribution in C-Prolog and Quintus sources, July 1988.

[19] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[20] Gopalan Nadathur and Dale Miller. An Overview of λProlog. In *Fifth International Logic Programming Conference*, pages 810–827, Seattle, Washington, August 1988. MIT Press.

[21] Gopalan Nadathur and Frank Pfenning. The type system of a higher-order logic programming language. In Frank Pfenning, editor, *Types in Logic Programming*, pages 245 – 283. MIT Press, 1992.