Higher Order Logic Programming with Untyped Lambda Expressions

James H. Andrews

Dept. of Computer Science
University of Western Ontario
London, Ontario, Canada N6A 5B7
andrews@csd.uwo.ca

Abstract. A higher order logic programming system is presented. The declarative semantics of the system is based on the type-free higher order logic NaDSyL, which takes a nominalist approach to solving the set-theoretic paradoxes. The operational semantics is based on the deterministic and useful subset of higher order unification known as pattern unification. It is shown that the system allows all expressions of the untyped lambda calculus, including the Y combinator and expressions capturing recursive functions, without losing consistency due to Curry's paradox. The system automatically performs lazy function application and thus unifies higher order logic and functional programming in a simple and elegant manner.

1 Introduction

The three notions of logic programming, functional programming, and higher order have been combined in various ways in recent years. We have higher-order functional languages like ML, higher order logic languages like Lambda Prolog, and proposals for combined functional-logic programming. All these systems are built at least in part on Church's type theory. But a fundamental semantic barrier – Curry's paradox – stands between these systems and an integrated higher order functional-logic language.

This paper shows that a language built on what is in many ways a simpler basis can also integrate higher order logic and functional programming. The semantic basis of the language is Paul Gilmore's "nominalistic" higher order logic NaDSyL [Gil97], which avoids the set-theoretic paradoxes by making a syntactic distinction between use and mention, without requiring a type system for consistency. The operational basis of the language is pattern unification, the deterministic and incomplete but nevertheless practically useful subset of higher order unification first described by Dale Miller [Mil91].

1.1 Curry's Paradox and Programming Languages

Researchers from Frege to Curry struggled with the problem of defining a logic which included both relational and functional abstraction, and yet remained consistent. Curry eventually boiled down the consistency problem to a paradox which applied to any language powerful to express both implication and a fixed-point combinator such as the Y combinator.

Hindley and Seldin [HS86] express Curry's paradox thus. Given a formula Z, we define X to be $Y(\lambda z.(z \rightarrow (z \rightarrow Z)))$, where Y is the Y combinator. By applying the usual properties of Y and the axioms of implication, we can eventually conclude that both X and Z are true. Unfortunately, Z is any arbitrary formula, so we can prove anything and our system is inconsistent.

Church's response to such paradoxes, of course, was to formulate his system T, disallowing the Y combinator and assigning a type to each term and formula [Chu40]. This has led to type theory, which forms the basis of the type system of both ML and Lambda Prolog. Cardelli [Car96], among others, shows that we can go further in the functional sphere by introducing *recursive* or *fixpoint* types, which supply a type to the Y combinator. But this does not extend to logic programming.

Combined functional-logic languages [AKP93, Han97] typically do not attempt to move to higher order. The farthest that has been gone so far is to allow lambda abstraction over terms in a functional-logic language [KA96], using higher order narrowing; but relational abstraction is forbidden.

1.2 Gilmore's Logics Applied to Programming

So far the focus has been on type theory as a solution to the paradoxes. But there are other solutions. Gilmore [Gil80, Gil86, Gil97] has built simple and consistent logical systems with general lambda abstraction over terms and formulas, which require no notion of type to maintain consistency (other than minimal type-like notions such as the order of a constant). Curry's paradox is avoided in these systems due to a relatively simple restriction on the form of the reflexive entailment axiom, $a \vdash a$.

This paper bases a higher order logic programming language, Nom, on the latest of these logics, NaDSyL [Gil97]. The immediate advantage of this is simplification: we do not need a type system just in order to define the underlying logic. But we get another, perhaps surprising, advantage: we now allow all expressions of the *untyped* lambda calculus as terms. Thus we have access to self-contained terms representing recursive functions, and thus to functional programming.

We of course want a type system in such a programming language, for reasons like validating our programs at compile time. When we choose Nom as a base language, we can choose our type system for expressivity from the rich assortment available to us [Car96], free of consistency considerations. However, this topic is beyond the scope of this paper.

The structure of the rest of this paper is as follows. In Section 2 we present the logic, based on NaDSyL, on which the programming language is based. In Section 3 we present the operational semantics of the language Nom, and a proof of its soundness. In Section 4, we present examples which illustrate that Nom retains much of the expressiveness of HOLP languages, is enriched with all untyped lambda calculus terms, and in fact automatically performs lazy evaluation on such terms. Section 5 discusses related results, and Section 6 presents some conclusions and directions for future work.

2 The Gilmore-Style Logic

In this section, we present the logical basis of Nom. We begin with some motivation, and then discuss a logic (presented proof-theoretically) in the Gilmore style. The logic is a simplified version of Gilmore's NaDSyL [Gil97], but is still powerful enough to form the basis of a HOLP system.

2.1 Motivation

Gilmore's "nominalist" view of the paradoxes is that they arise from a confusion between use and mention. Second order quantification variables represent the use of a relation, but every relation is amongst only the *names* of things, that we are merely mentioning in that context. Thus, second order quantification variables cannot be arguments of other relations, although second order constants and individual terms without free second order variables can.

The syntactic notion that this leads to is here called *Gilmore atomicity*: a formula $(e\ f_1\ f_2\ \ldots\ f_n)$ is Gilmore atomic if e is a second order constant or variable, and no f_i contains any free second order variable. In Gilmore's system NaDSyL [Gil97], the formulas that can appear as the key formula a in the reflexive entailment axiom

$$\overline{\Gamma, a \vdash \Delta, a}$$

are just the Gilmore atomic formulas. The rest of the rules of NaDSyL are an extremely straightforward extension of first order logic to set abstraction. The Gilmore atomicity restriction is sufficient to achieve consistency.

2.2 A Useful Subsystem of NaDSyL

Here we present the syntax and proof theory of a subsystem of NaDSyL which will form the basis of Nom. The main differences from the system in [Gil97] are that the syntax is simplified to a slight generalization of the lambda calculus; the two connectives of conjunction and negation are used instead of the single joint denial connective; and the condition on the $\forall$/l rule is simplified, making the system weaker than the original NaDSyL. This resultant weakness is inessential for the present purposes, but buys us the advantage of simplicity. Space does not permit a proof of the consistency of the system, but it follows along the same lines as in [Gil97].

We assume the existence of countable sets of first order variables, second order variables, first order constants and second order constants. The syntax of expressions E can be expressed in BNF as
$E\ ::=\ X_1\ \mid\ X_2\ \mid\ C\ \mid\ Q\ \mid\ (EE)\ \mid\ \lambda X_1.E$

where X_1 is a first order variable; X_2 is a second order variable; C is a first or second order constant; and Q is one of the connectives *and*, *not*, $forall_1$, or $forall_2$ (the last two connectives representing first and second order quantification).

We use a, b, c as metavariables standing for first order constants, and p as a metavariable standing standing for a second order constant (i.e., a predicate). We use e, f, g, s, t as metavariables standing for arbitrary expressions, and x, y, z as metavariables standing for variables[1]. The order of a variable will be apparent from context.

As is standard, we write the expression $(\ldots((e_1\ e_2)e_3)\ldots e_n)$ as $(e_1\ e_2\ \ldots\ e_n)$. We write $e\&f$, $\neg e$, and $\forall_i x.e$ for $(and\ e\ f)$, $(not\ e)$, and $forall_i\ \lambda x.f$, respectively. We define α, β and *η-convertibility* in the usual way (treating connectives as if they were constants). Two expressions are *$\alpha\beta\eta$-equivalent* if they are convertible to the same expression via an arbitrary number of α, β, or η-conversion steps.

A *basic-mention* expression is one in which no free second order variables appear. A *basic-use* expression is a second order constant or variable. A *Gilmore atomic* expression is one of the form $(e\ f_1\ f_2\ \ldots\ f_k)$, $k \geq 0$, where e is a basic-use expression and each f_i is a basic-mention expression.

A *sequent* is an ordered pair of finite sets of expressions, written in the form $\Gamma \vdash \Delta$. We will use Γ and Δ as metavariables standing for sets of expressions; the notation Γ, Δ will mean $\Gamma \cup \Delta$, and Γ, e will mean $\Gamma \cup \{e\}$. We present the proof system in the form of Gentzen-style proof rules for sequents in Figure 1.

In Gilmore's original presentation [Gil97], he defines a semi-decidable set of *formulas*; the restriction on the $\forall$/l rule then essentially says that the upper expression must be a formula. The condition here is weaker, but simplifies the presentation considerably.

Since the rules define a complete set of classical connectives, it is clear that we can introduce all the other classical connectives, such as $\vee$ (disjunction), $\rightarrow$ (implication) and $\exists$ (existential quantifier). We will use these connectives in the sequel without further comment.

3 The Logic Programming Language

3.1 Syntax and Notation

The syntax of Nom is the same as that of NaDSyL, except that first and second-order *placeholder variables* are allowed. These variables are used in the operational semantics of Nom, and correspond to the uninstantiated variables of Prolog. They are substituted for existentially quantified variables in goals, as placeholders for the terms to be discovered later by unification[2]. They are distinct from the regular, logical variables mentioned in the last section.

[1] This usage unfortunately but necessarily clashes with Gilmore's original notation, in which upper case denotes second order and lower case denotes first order.

[2] Conventional logic programming systems call these "free variables", but here that convention is misleading since we have regular variables which can appear either free or bound. Nipkow [Nip93] calls free, bound and placeholder variables respectively

Reflexive entailment: $$\overline{\Gamma, e \vdash \Delta, e}$$

where e is Gilmore atomic

&/l: $$\frac{\Gamma, e, f \vdash \Delta}{\Gamma, e\&f \vdash \Delta}$$ &/r: $$\frac{\Gamma \vdash \Delta, e \quad \Gamma \vdash \Delta, f}{\Gamma \vdash \Delta, e\&f}$$

$\neg$/l: $$\frac{\Gamma \vdash \Delta, e}{\Gamma, \neg e \vdash \Delta}$$ $\neg$/r: $$\frac{\Gamma, e \vdash \Delta}{\Gamma \vdash \Delta, \neg e}$$

$\forall$/l: $$\frac{\Gamma, e[x := f] \vdash \Delta}{\Gamma, \forall_i x(e) \vdash \Delta}$$

where $i = 1$ and f is basic-mention, or $i = 2$ and f is basic-use

$\forall$/r: $$\frac{\Gamma \vdash \Delta, e[x := y]}{\Gamma \vdash \Delta, \forall_i x(e)}$$

where y is a variable of order i not appearing in Γ, Δ, e

β/l: $$\frac{\Gamma, (e[x := f_1]\ f_2\ \ldots\ f_n) \vdash \Delta}{\Gamma, ((\lambda x.e)\ f_1\ f_2\ \ldots\ f_n) \vdash \Delta}$$ β/r: $$\frac{\Gamma \vdash (e[x := f_1]\ f_2\ \ldots\ f_n), \Delta}{\Gamma \vdash ((\lambda x.e)\ f_1\ f_2\ \ldots\ f_n), \Delta}$$

Conv/l: $$\frac{\Gamma, e' \vdash \Delta}{\Gamma, e \vdash \Delta}$$ Conv/r: $$\frac{\Gamma \vdash e', \Delta}{\Gamma \vdash e, \Delta}$$

where e and e' are Gilmore atomic and $\alpha\beta\eta$-equivalent

Fig. 1. Proof rules of the subsystem of NaDSyL.

We use the upper case letters X, Y, Z to stand for these placeholder variables. We extend the notion of basic-mention expressions to allow first order placeholder variables, and basic-use expressions to allow second order placeholder variables.

We define a *definition* formula as a closed (ground) expression of the form $(p\ t_1\ \ldots\ t_n)$, of the form $(p\ t_1\ \ldots\ t_n) \leftarrow f$, or of the form $\forall_1 x(e)$, where e is a definition.

3.2 Pattern Unification and Safe Substitutions

The basis for the operational semantics of Nom is *pattern unification*, and we further require that the substitutions returned by pattern unification be *safe*.

Nipkow [Nip93] defines a *pattern* as a term t in β-normal form such that every occurrence of a placeholder variable X is in a subterm $(X\ y_1\ \ldots\ y_n)$, where

"loose bound", "bound" and "free" variables. However, because we are relating operational semantics to conventional logic we must adopt a different nomenclature.

the y_is are η-equivalent to distinct regular variables. Pattern unification, first discovered and applied to Lambda Prolog by Miller [Mil91], is higher order unification restricted to patterns. Although not all lambda-expressions are patterns, in practice patterns come up frequently. Pattern unification returns at most one solution, and so can be used as a "basic engine" of a logic programming language, much as first order unification can for Prolog.

For a higher order logic programming language with universal quantification, we need another restriction, as discussed by Nadathur et al. [NJW93, NJK94]. Placeholder variables are intended to stand for eigenvalues, so they cannot be substituted for by regular or placeholder variables introduced later in the computation; otherwise the side-condition on the $\forall/r$ rule could be implicitly circumvented. Hence we assume that all new regular and placeholder variables are tagged internally with a number representing the stage in the computation at which they were introduced.

Finally, for a language based on NaDSyL, we need a second restriction. First and second order placeholder variables are intended to stand for first and second order eigenvalues, and thus must obey those restrictions. Hence we say that a substitution is *safe* if no X is substituted by a term containing a newer variable, if each first order X is substituted by a basic-mention term, and if each second order X is substituted by a basic-use term.

3.3 Operational Semantics

The basic elements of the operational semantics for Nom are sequences of sequents, enriched with placeholders. To pose the query f in the context of the definition formulas $e_1, \ldots, e_n$, we form the sequence consisting of the single sequent $(e_1, \ldots, e_n \vdash f)$, and begin computation. Each computation rule in the operational semantics takes one sequence to another; when we reach the empty sequence of sequents, the computation has terminated successfully. The semantics is nondeterministic. A query is successful if it has a successful computation path, but for simplicity we do not discuss the search for such a path.

The operational semantics for Nom are given in Figure 2. It follows conventional lines, e.g. from [Der89, NM94], with the necessary extensions for iterated implication and higher order features. We use Γ to stand for a sequence of definition formulas, and α to stand for a sequence of sequents (generally representing the remaining sequents in the query). $\overline{s}$ and $\overline{t}$ are sequences of terms. Substitutions here are substitutions of terms for placeholder variables, and are done using pattern unification.

We will discuss examples extensively in the next section, but for now let us prove the soundness of the operational semantics with respect to the original proof system.

Theorem 1. Let α be a sequence of sequents such that $\alpha \Rightarrow^* \epsilon$, where ϵ is the empty sequence. Then there is some safe substitution ρ such that for every sequent S in α, $S\rho$ is derivable.

Right-hand rules:
R1: $(\Gamma \vdash e\&f), \alpha \quad \Rightarrow (\Gamma \vdash e), (\Gamma \vdash f), \alpha$
R2: $(\Gamma \vdash e \vee f), \alpha \quad \Rightarrow (\Gamma \vdash e), \alpha$
R3: $(\Gamma \vdash e \vee f), \alpha \quad \Rightarrow (\Gamma \vdash f), \alpha$
R4: $(\Gamma \vdash e \rightarrow f), \alpha \quad \Rightarrow (\Gamma, e \vdash f), \alpha$
where e is a definition
R5: $(\Gamma \vdash (\lambda x.e\ f)), \alpha \Rightarrow (\Gamma \vdash e[x := f]), \alpha$
R6: $(\Gamma \vdash \exists_n x(e)), \alpha \quad \Rightarrow (\Gamma \vdash e[x := X]), \alpha$
where X is a new nth order placeholder variable
R7: $(\Gamma \vdash \forall_n x(e)), \alpha \quad \Rightarrow (\Gamma \vdash e[x := y]), \alpha$
where y is a new nth order regular variable

Left-hand rules:
L1: $(\Gamma, e \vdash f), \alpha \quad \Rightarrow (\Gamma, e, e \vdash f), \alpha$
L2: $(\Gamma, \forall_n x(e) \vdash f), \alpha \quad \Rightarrow (\Gamma, e[x := X] \vdash f), \alpha$
where X is a new nth order placeholder variable
L3: $(\Gamma, ((p\ \overline{s}) \leftarrow g) \vdash p\ \overline{t}), \alpha \Rightarrow (\Gamma\theta \vdash g\theta), \alpha\theta$
where $(p\ \overline{s})$ and $(p\ \overline{t})$ are Gilmore atomic, and
where θ is the pattern unifier of $\overline{s}$ and $\overline{t}$, and θ is safe
L4: $(\Gamma, (p\ \overline{s}) \vdash p\ \overline{t}), \alpha \quad \Rightarrow \alpha\theta$
where $(p\ \overline{s})$ and $(p\ \overline{t})$ are Gilmore atomic, and
where θ is the pattern unifier of $\overline{s}$ and $\overline{t}$, and θ is safe

Fig. 2. Operational semantics for the programming language Nom.

Proof. The proof proceeds by induction on the number of steps in the computation. The base case is trivial. The induction case has subcases on the form of rule applied in the first step of the computation. The subcases for most of the rules are straightforward; we will concentrate on R6, R7 and L4.

R6: By the induction hypothesis, there is some safe ρ such that $(\Gamma\rho \vdash e[x := X]\rho)$ and $\alpha\rho$ are derivable. But then $(\Gamma\rho \vdash e[x := t]\rho)$, where t is $X\rho$, is also derivable; and hence $(\Gamma\rho \vdash (\exists_n x(e))\rho)$ is as well. Hence ρ is the desired substitution.

R7: By the induction hypothesis, there is some safe ρ such that $(\Gamma\rho \vdash e[x := y]\rho)$ and $\alpha\rho$ are derivable. But since ρ is safe, and all placeholder variables in Γ and e have been created before y, y does not appear free in either $\Gamma\rho$ or $e\rho$. Hence $(\Gamma\rho \vdash (\forall_n x(e))\rho)$ is also derivable. Hence ρ again is the desired substitution.

L4: By the induction hypothesis, there is some safe ρ' such that the $\alpha\theta\rho'$ sequents are derivable. By the properties of unifiers, we know that the $\overline{s}\theta$ terms are $\alpha\beta\eta$-equivalent to the $\overline{t}\theta$ terms, and by the Gilmore atomicity side-condition and the safety side-condition, we know that $(p\ \overline{s})\theta$ and $(p\ \overline{t})\theta$ are both Gilmore atomic. Hence $(\Gamma\theta, (p\ \overline{s})\theta \vdash (p\ \overline{t})\theta)$ is derivable. But then setting $\rho = \theta\rho'$, we have that both $(\Gamma\rho, (p\ \overline{s})\rho \vdash (p\ \overline{t})\rho)$ and the $\alpha\rho$ sequents are also derivable. Hence ρ is the desired substitution. Q.E.D.

4 Examples

This section presents three examples. The first shows that local declarations of relations are possible in Nom, as they are in Lambda Prolog. The second shows that predicates and lambda expressions representing predicates can be passed as parameters, but not second order variables; we argue that the restriction is not important. The third shows that we can use general untyped lambda calculus terms in Nom, including terms representing infinite lists, which are automatically evaluated in a lazy manner.

In these examples, we generally compute by choosing the appropriate R rules until we reach an atomic formula on the right of the turnstile, and then choose L rules which act on definitions of the predicate in that atomic formula.

4.1 Local Relation Declarations

Lambda Prolog allows universal quantification over relations in queries and implications of goals by definitions, hence allowing a kind of "local declaration" of relations [NM94]. Here we show that Nom has this capability as well.

Let *rev* be a constant and *revapp* be a variable, both second order. Let
$\mathbf{C}_{11} = \forall_1 y(revapp\ [\,]\ y\ y)$,
$\mathbf{C}_{12} = \forall_1 x, xs, yx, zs((revapp\ (x :: xs)\ ys\ zs) \leftarrow (revapp\ xs\ (x :: ys)\ zs))$,
$\mathbf{C}_{13} = \forall_1 xs, zs((rev\ xs\ zs) \leftarrow \forall_2 revapp(\mathbf{C}_{11} \rightarrow \mathbf{C}_{12} \rightarrow (revapp\ xs\ [\,]\ zs)))$.
Let the sequence α_1 of sequents be the single sequent $(\mathbf{C}_{13} \vdash (rev\ [a]\ Zs))$, where Zs is a first order placeholder. If we proceed with computation from α_1, some of the resulting steps are the following.

- $(\mathbf{C}_{13} \vdash \forall_2 revapp(\mathbf{C}_{11} \rightarrow \mathbf{C}_{12} \rightarrow (revapp\ [a]\ [\,]\ Zs)))$. This arises after a conventional logic programming predicate application sequence.
- $(\mathbf{C}_{13}, \mathbf{C}'_{11}, \mathbf{C}'_{12} \vdash (revapp'\ [a]\ [\,]\ Zs)))$,
  where $revapp'$ is a new second order variable, and $\mathbf{C}'_{11}, \mathbf{C}'_{12}$ are $\mathbf{C}_{11}, \mathbf{C}_{12}$ with $revapp$ substituted by $revapp'$.
- $(\mathbf{C}_{13}, \mathbf{C}'_{11}, \mathbf{C}'_{12}, ((revapp'\ (X :: Xs)\ Ys\ Zs') \leftarrow$
  $(revapp'\ Xs\ (X :: Ys)\ Zs')) \vdash (revapp'\ [a]\ [\,]\ Zs)))$,
  where X, Xs, Ys, Zs' are all new first order placeholders. At this point, all the indicated atoms are Gilmore atomic even though there is a second order variable involved, and there is a safe first order unifier (and thus pattern unifier) of the goal on the right of the $\vdash$ with the head of the clause on the left.
- $(\mathbf{C}_{13}, \mathbf{C}'_{11}, \mathbf{C}'_{12}, (revapp\ [\,]\ Y\ Y) \vdash (revapp'\ [\,]\ [a]\ Zs)))$,
  where Y is a new first order placeholder. At this point, pattern unification gives us a substitution of both Y and Zs by $[a]$, so by applying L4 we obtain
- ϵ, the empty sequence.

Hence the computation succeeds with the correct substitution of Zs by $[a]$.

4.2 Relations as Parameters

Here we study the passing of relations and relation-like terms to other relations, using the classic program *mapr* which maps a binary predicate down a list.

Let *mapr* and *eq* be second order constants. Let
$\mathbf{C}_{21} = \forall_1 r(mapr\ r\ [\,]\ [\,])$,
$\mathbf{C}_{22} = \forall_1 r, x, xs, y, ys((mapr\ r(x :: xs)(y :: ys)) \leftarrow (r\ x\ y)\&(mapr\ r\ xs\ ys))$,
$\mathbf{C}_{23} = \forall_1 x(eq\ x\ x)$.
Let $\mathbf{P}$ be the expression $\lambda x.\lambda y.(eq\ y\ (pair\ x\ x))$, where *pair* is a first order constant. Finally, let the sequence α_2 of sequents be the single sequent $(\mathbf{C}_{21}, \mathbf{C}_{22}, \mathbf{C}_{23} \vdash (mapr\ \mathbf{P}\ [a, b]\ Ys)$, where Ys is a first order placeholder. Some of the resulting computation stages are the following.

- $(\mathbf{C}_{21}, \mathbf{C}_{22}, \mathbf{C}_{23}, ((mapr\ R\ (X :: Xs)\ (Y :: Ys')) \leftarrow$
  $(R\ X\ Y)\&(mapr\ R\ Xs\ Ys')) \vdash (mapr\ \mathbf{P}\ [a, b]\ Ys)$.
  Again, the goal and the head of the clause are Gilmore atomic, so unification proceeds normally, unifying Ys with $(Y :: Ys')$. After another step we obtain
- $(\mathbf{C}_{21}, \mathbf{C}_{22}, \mathbf{C}_{23} \vdash (\mathbf{P}\ a\ Y)), (\mathbf{C}_{21}, \mathbf{C}_{22}, \mathbf{C}_{23} \vdash (mapr\ \mathbf{P}\ [b]\ Ys'))$,
  our first sequence with more than one sequent, due to the conjunction in the definition of *mapr*. After beta-reduction of the goal and more steps to the left of the turnstile, we obtain
- $(\mathbf{C}_{21}, \mathbf{C}_{22}, \mathbf{C}_{23}, (eq\ X\ X) \vdash (eq\ Y\ (pair\ a\ a)))$,
  $(\mathbf{C}_{21}, \mathbf{C}_{22}, \mathbf{C}_{23} \vdash (mapr\ \mathbf{P}\ [b]\ Ys'))$.
  Y here is unified with $(pair\ a\ a)$, resulting in
- $(\mathbf{C}_{21}, \mathbf{C}_{22}, \mathbf{C}_{23} \vdash (mapr\ \mathbf{P}\ [b]\ Ys'))$.

The computation continues for a few more steps; ultimately the final value of Ys is $[(pair\ a\ a), (pair\ b\ b)]$, as desired.

Note that this computation would proceed normally with any binary second order constant in the place of $\mathbf{P}$. However, it would *not* succeed with a second order *variable* (or placeholder) in the place of $\mathbf{P}$. This is because the goal and the clause head in the first computation state above would not be Gilmore atomic.

This curious restriction would be troubling in Lambda Prolog, although not onerous. (None of the examples we have found in the Lambda Prolog literature would run afoul of it.) However, in Nom, we have general *functional* abstraction, including recursive function terms (see next example); hence we can still pass recursive constructs as arguments, even if locally-defined recursive predicates are not allowed.

As Hanus points out [Han97], many of the relations in logic programs are actually functions, and so would more naturally be expressed by the recursive function terms of Nom. We therefore do not expect that this Gilmore atomicity restriction would unduly affect the practical use of languages like Nom.

4.3 Recursive Function Terms

Having access to the Y combinator [HS86] gives us the ability to express any recursive function of the untyped lambda calculus. Here we look at how Nom

computes with such terms. Let
$\mathbf{V} = \lambda y.(x\ (y\ y))$,
$\mathbf{Y} = \lambda x.(\mathbf{V}\ \mathbf{V})$.
This is Curry's definition of the Y combinator. Further let
$\mathbf{Ints} = \mathbf{Y}(\lambda ints.\lambda x.(x :: (ints\ (s\ x))))$.
The term ($\mathbf{Ints}\ 0$) represents an infinite list of the integers in Peano notation, i.e. $[0, (s\ 0), (s\ (s\ 0)), \ldots]$. Clearly this term cannot be computed completely; and yet it is a valid term of Nom, is a pattern (since it contains no placeholders), and is even basic-mention (since it contains no second order variables).

Now let
$\mathbf{C}_{31} = \forall_1 x, xs(everythird\ (x :: xs)\ x)$,
$\mathbf{C}_{32} = \forall_1 x_1, x_2, x_3, xs, x((everythird\ (x_1 :: x_2 :: x_3 :: xs)\ x) \leftarrow (everythird\ xs\ x))$.
$(everythird\ x\ y)$ therefore holds if y is one of every third element of x.

Let the sequence α_3 of sequents be the single sequent $(\mathbf{C}_{31}, \mathbf{C}_{32}, \mathbf{C}_{33} \vdash (everythird\ (\mathbf{Ints}\ 0)\ Y))$, where Y is a first order placeholder. The computation can proceed, in part, as follows:

- $(\mathbf{C}_{31}, \mathbf{C}_{32}, \mathbf{C}_{33}, (everythird\ (X :: Xs)\ X) \vdash (everythird\ (\mathbf{Ints}\ 0)\ Y)$, where X and Xs are new first order placeholders. At this point, Nipkow's version of pattern unification [Nip93] automatically reduces the term ($\mathbf{Ints}\ 0$) to head-normal form, that is to $(0 :: (\mathbf{Ints}\ (s\ 0)))$. We therefore obtain a substitution of Y by 0, and the next step is
- ϵ, the empty sequence, indicating success.

Other choices of computation path will result in a unification of ($\mathbf{Ints}\ 0$) with a term of the form $(X_1 :: X_2 :: X_3 :: Xs)$. This will result in pattern unification repeatedly finding the head-normal form of an **Ints** term, with the effect that it adds more and more evaluated elements to the list.

Hence the basic mechanism of pattern unification, when applied to Nom, results in lazy evaluation of recursive function expressions. This is clearly not restricted to the example, but applies to every recursive function. We can therefore take a function definition of the form $\mathbf{f}x = e(\mathbf{f})$ as meaning that the term $\mathbf{f}$, wherever it is used, stands for the term $(\mathbf{Y}\ \lambda f.\lambda x.e(f))$. This approach to function definition, reminiscent of the semantics of Lisp, lets Nom encompass functional programming as well as higher order logic programming.

5 Related Work

This paper can be seen as a successor to an earlier paper of the author [And89], which gave a simpler language based on the less expressive logic NaDSet [Gil86].

Besides the other work cited in the text, this paper is also related to the work of Chen, Kifer and Warren on HiLog [CKW89], a language with a first-order semantics in which quantified variables range only over the constants defined in the program [And89]. Finally, it is related to the work of Kamareddine [Kam92a, Kam92b], who gives a particular set of rules and a type system for higher order logic which permit the typing of the Y combinator, but which do not permit general lambda terms.

6 Conclusions and Future Work

We have described a higher order logic programming system based on Gilmore's logic NaDSyL. The language has most of the essential power of Lambda Prolog and can be implemented in much the same manner, but also has general untyped lambda calculus terms which it evaluates in a lazy manner. This gives it the power of functional programming as well.

Topics for future research include:

- Implementation of the language; this may best be done by modifying an existing Lambda Prolog implementation.
- Adding a type system to the language. It seems that some Lambda Prolog-type system with fixpoint types might be good enough, but we have considerable latitude, since types are not needed for consistency.
- Adding some form of negation. The problem of negation has not even been fully resolved in first order logic programming, and has never been addressed in HOLP to our knowledge. Hence we must proceed carefully here.

7 Acknowledgments

Thanks to Paul Gilmore for many invaluable conversations about this work. Thanks also to Peter Apostoli, Bharat Jayaraman, and George Tsiknis for important observations. This work was supported by Paul Gilmore's NSERC operating grant and by the FormalWare project, an initiative headed by Jeff Joyce and funded by the BC Advanced Systems Institute, Hughes Aircraft Canada Ltd., and Macdonald Dettwiler.

References

[AKP93] Hassan Ait-Kaci and Andreas Podelski. Towards a meaning of Life. *Journal of Logic Programming*, 16(3/4):195, July 1993.

[And89] James H. Andrews. Predicates as parameters in logic programming: A set-theoretic basis. In *Proceedings of Workshop on Extensions to Logic Programming*, volume 475 of *Lecture Notes in Artificial Intelligence*, pages 31–47, Tübingen, Germany, December 1989. Springer.

[Car96] Luca Cardelli. Type systems. In *CRC Handbook of Computer Science and Engineering*. CRC Press, 1996.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[CKW89] Weidong Chen, Michael Kifer, and David S. Warren. HiLog: A first-order semantics of higher-order logic programming constructs. In *Proceedings of the North American Conference on Logic Programming*, Cleveland, Ohio, October 1989.

[Der89] Pierre Deransart. Proofs of declarative properties of logic programs. In *Theory and Practice of Software Engineering*, volume 351 of *Lecture Notes in Computer Science*, pages 207–226, Barcelona, Spain, 1989. Springer-Verlag.

[Gil80] Paul C. Gilmore. Combining unrestricted abstraction with universal quantification. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 99–123. Academic Press, 1980.

[Gil86] Paul C. Gilmore. Natural deduction based set theories: A new resolution of the old paradoxes. *Journal of Symbolic Logic*, 51(2):393–411, June 1986.

[Gil97] Paul C. Gilmore. NaDSyL and some applications. In *Proceedings of the Kurt Gödel Symposium*, Vienna, 1997. Available from UBC Computer Science as Technical Report 97-1.

[Han97] Michael Hanus. A unified computation model for functional and logic programming. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 80–93, Paris, 1997.

[HS86] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and Lambda Calculus.* Number 1 in London Mathematical Society Student Texts. Cambridge University Press, Cambridge, 1986.

[KA96] Herbert Kuchen and Josef Anastasiadis. Higher order babel: Language and implementation. In *Extensions of Logic Programming*, volume 1050 of *LNCS*, pages 193–207, Leipzig, Germany, March 1996. Springer.

[Kam92a] Fairouz Kamareddine. Lambda-terms, logic, determiners and quantifiers. *Journal of Logic, Language and Information*, 1(1):79–103, 1992.

[Kam92b] Fairouz Kamareddine. A system at the cross-roads of functional and logic programming. *Science of Computer Programming*, 19:239–279, 1992.

[Mil91] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. In Peter Schroeder-Heister, editor, *Extensions of Logic Programming*, number 475 in LNCS, pages 253–281. Springer, 1991.

[Nip93] Tobias Nipkow. Functional unification of higher-order patterns. In *Proceedings of the 8th IEEE Symposium on Logic in Computer Science (LICS)*, pages 64–74, 1993.

[NJK94] Gopalan Nadathur, Bharat Jayaraman, and Keehang Kwon. Scoping constructs in logic programming: Implementation problems and their solution. Technical Report CS-1994-35, Dept. of Computer Science, Duke University, Durham, NC, October 1994.

[NJW93] Gopalan Nadathur, Bharat Jayaraman, and Debra Sue Wilson. Implementation considerations for higher-order features in logic programming. Technical Report CS-1993-16, Dept. of Computer Science, Duke University, Durham, NC, June 1993.

[NM94] Gopalan Nadathur and Dale Miller. Higher-order logic programming. In D. Gabbay, C. Hogger, and A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, Oxford, 1994. Oxford University Press.