Programming in Higher-Order Logic

Alwen Tiu
Logic and Computation Group
College of Engineering and Computer Science
The Australian National University

December 12, 2011

1 Introduction

In this course we examine the use of logic as a programming language. In this programming paradigm, a program is usually represented as a set of formulae (typically, first-order formulae) and computation is specified as *search* for a particular consequence (also called a *goal*) of this set of formula. Roughly speaking, this is nothing more than theorem proving, i.e., deriving a consequence from a logical theory, albeit in a rather specific way. For example, the programming language Prolog is based on a fragment of classical first-order logic, and its underlying proof search mechanism is based on (first-order) resolution.

As a programming language, pure Prolog lacks many important constructs that often make it inadequate for sophisticated programming tasks. This ranges from low-level issues such as support for networking, I/O, etc., to the more conceptual level issues such as a (logical) notion of *control*, data types, modules and higher-order programming. Numerous attempts have been made to extend Prolog in various ways to incorporate one feature or another, and in some of these extensions, the connection to logic is somewhat lost, or at best, marginalised. In this lecture, we shall look at those extensions which are *purely logical*. In particular, we shall look at an extension of Prolog, called λProlog, which incorporates, to some degrees, notions such as higher-order programming, modules and abstract data types.

In the search for purely logical extensions of Prolog, a group of researchers was led to re-examining one fundamental question: what really is logic programming? This re-examing has lead to the identification of logic programming with *goal-directed proof search*. More specifically, logical connectives should be seen as *instructions* for proof search. To elaborate more on this point, suppose Γ represents a program (which is just a set of formula). We could then "execute" the program by specifying a goal, say, $A \wedge B$, which amounts to asking the language interpreter to find a proof of $\Gamma \vdash A \wedge B$. When one is concerned only with proof search, one could view the conjunction $A \wedge B$ as an instruction of search: prove A *and* prove B. Similarly, to prove a goal $A \vee B$ in a goal-directed manner means we have to prove A *or* B. The important foundational question is whether such a goal-directed search strategy is *complete*, i.e., whether any valid consequence relation can be derived using this search strategy. Then a logical system can be seen as a logic programming language if goal-directed search in

the logical system is complete. This view of logic programming as goal-directed proof search is often called *abstract logic programming.*

Note that goal directed proof search may not be complete for some (fragments of) logics. For example, if we take classical propositional logic as the underlying semantics and classical propositional formula as logic programs, then there are some valid consequence relations that are not derivable via goal-directed search strategy. Consider a program $\mathcal{P} = \{p \vee (q \wedge r)\}$ and a goal $G = p \vee q$. It is easy to see that $\mathcal{P} \vdash G$ is valid classically. However, if we apply a goal-directed search for $\mathcal{P} \vdash G$, then we end up having to prove $\mathcal{P} \vdash p$ or $\mathcal{P} \vdash q$. But neither one of them is valid. In this course, we shall use *proof theory* as the foundation for logic programming, instead of the more traditional approach based on model theory (e.g., as in pure Prolog). In particular, we shall use sequent calculus as the formalism for presenting proof systems. Sequent calculus is a natural choice in this case, since we base our extended logic programming language on proof search, and sequent calculus is perhaps the most natural formalism for this purpose. We shall be using a non-classical logic, called intuitionistic logic, as the foundation for the extensions of logic programming discussed here. This is in contrast to Prolog, which is based on first-order classical logic. As we shall see, there is an intrinsic property of classical logic that makes it difficult to accomodate a logical notion of modular programming in a straightforward and purely logical fashion.

Outline of the course This course consists of five lectures. A brief description of each of the lectures follows:

- Lecture 1: Intuitionistic logic. In this lecture, we cover the proof theoretic foundation which will be used in subsequent lectures. A sequent calculus of intutionistic logic is introduced. We also present a representation language for encoding various syntactic structures using simply typed λ-calculus. An overview of some basic notions related to λ-calculus is presented.

- Lecture 2: Abstract logic programming. This lecture introduces the abstract notion of a logic programming language. We then show an instance which is based on the Horn-fragment of first-order intuitionistic logic. Completeness of goal-directed proof search is proved.

- Lecture 3: Higher-order Horn clauses. Continuing on the topic of the previous lecture, we discuss an important algorithm which forms the core of implementations of logic programming language: the unification algorithm (for the first-order case). We then give an overview of the λProlog language, and show some example logic programs written in λProlog. This concludes the first part of the course on first-order Horn clauses.

  We then present the higher-order extension of Horn clauses, by allowing quantification over predicates. The proof of the completeness of goal directed proof search is discussed.

- Lecture 4: Higher-order programming: We first present the unification algorithm for the higher-order case, i.e., Huet's algorithm for simply typed λ-calculus. This is then followed by some examples (in λProlog) illustrating the use of higher-order quantification.

- Lecture 5: Hereditary Harrop Formulae. We now consider a further extension of higher-order Horn clauses by allowing implication in goal formulae. This class of formulae is called hereditary Harrop formulae. We show how the combination of predicate quantification and implication can be used to hide the internal representation of some predicates, thus allowing a simple notion of modular programming. We end this lecture by presenting an example of encoding a simple functional programming language, which demonstrates the crucial use of implication and universal quantification in goal formulae.

In the remainder of these notes, we briefly discuss some basic notions and definitions used throughout the course, e.g., proof rules, etc. Some bibliography notes are given in Section 7 for those who are interested in in-depth study of abstract logic programming. Slides for the course are available from the following URL:

`http://users.rsise.anu.edu.au/~tiu/teaching/lss09/`

The same website also contains λProlog codes that are used in this course. These codes have been tested with the Teyjus compiler for λProlog, available from: `http://teyjus.cs.umn.edu/` Students are encouraged to download the codes and the Teyjus system to experiment with the programs.

2 Intuitionistic logic

Inadequacy of classical logic As mentioned in the introduction, classical logic may not be the most suitable logic to capture notions such as modular programming. A *module* is usually characterised by the presence of some internal structures used exclusive in that module, which are not available to "external" modules or programs. A module can also be invoked by another module, e.g., the `include` directive in C, etc. Typically, in a modular language, access to functions or methods in a module is only available with explicit invocation or inclusion. To capture these characteristics logically, natural candidates would be a (universal/existential) quantifier and logical implication. Indeed, method invocation has an implication-like reading, i.e., a module B which uses A can be interpreted as "proving B given the assumption A". In this reading, the "scope" of the module A is only over B. However, in classical logic, we have the following tautology:

$$(p \supset q) \vee r \equiv p \supset (q \vee r).$$

So classical implication does not naturally capture the scoping aspect of a module. This phenomena, called "scope extrusion", also appears in classical quantifier:

$$(\forall x.P) \vee Q \equiv \forall x(P \vee Q)$$

provided x is not free in Q. Here x can be "accessed" by Q even though staticly it is not in the scope of x. Thus classical universal quantification does not encapsulate the above notion of data abstraction either.

Given the above inadequacy of classical logic, we turn to a different logic in which the scope extrusion laws do not necessarily hold in general. We shall use intuitionistic logic as the foundation for the extension of Prolog covered in this course. This particular choice of intuitionistic logic is partly because it

has been the most widely studied non-classical logic, and perhaps semantically more intuitive than other substructural logics. As we shall see in Section 7, other "constructive" logics can also be used as a foundation for abstract logic programming.

Intuitionistic logic is one of many constructive logics which have roots in the constructivist revision of foundations of mathematics in the end of the 19th century. Among the distinctive characters of intuitionistic logic are the rejection of the principle of "excluded middle" (i.e., A or not A is true) and the insistence on the constructive existence. The latter means that a proof of an existence of an object should give us the means of constructing the said object. This is in contrast to classical principle where the existence proof can sometimes mean freedom from contradiction.

2.1 Sequent calculus

Intuitionistic logic is often presented as a natural deduction proof system. We shall, however, use a sequent calculus presentation. This is because sequent calculus is a more suitable calculus for proof search. In traditional sequent calculus, such as Gentzen's LK and LJ, sequents usually take the form $\Gamma \longrightarrow \Delta$ where Γ and Δ are either sets or multisets of formulas. These formulas can contain free variables (also called eigenvariables, which are essentially scoped constants). Also, in Gentzen's LK or LJ, object level terms are relatively simple (they are just individual constants). In using sequent calculus to model computation, it is important that we can capture various syntactic structures of programming languages used to describe data structures, etc. These are usually encoded as object level terms in sequent calculus. Throughout this lecture, we shall augment sequent calculus with a rich term language, namely, the simply typed λ-calculus of Church. It can be shown that most programming constructs can be encoded as simply typed λ-terms, via the so-called higher-order abstract syntax (HOAS) encoding technique.

The language of λ-terms is generated from the following grammar:

$$s, t ::= x \mid (s\ t) \mid \lambda x.t$$

The term $(s\ t)$ is called an *application* and the term $\lambda x.t$ is called an *abstraction*. The variable x next to the λ symbol in $\lambda x.t$ acts as a binder. A variable occurrence in a term t is *free* if it is not under the scope of a λx. An important operation on λ-terms is *substitution* of free variables in the terms with other terms. This notion of substitution has to avoid "capture" of free variables. That is, a free variable in the range of a substitution should remain free in the substituted term. The set of free variables in a term t is denoted by $FV(t)$. This notation extends straightforwardly to sets of terms, e.g., if S is a set of terms then $FV(S)$ represents the union of $FV(t)$ for every $t \in S$.

A *substitution* θ is a mapping from variables to terms such that the *domain* of θ, i.e., the set $\{x \mid \theta(x) \neq x\}$ is finite. We denote with $dom(\theta)$ the domain of θ, and $ran(\theta)$ its range, i.e.,

$$ran(\theta) = \{\theta(x) \mid x \in dom(\theta)\}.$$

We often present a substitution as a list enclosed in brackets, e.g.,

$$[x_1 \mapsto t_1, \ldots, x_n \mapsto t_n].$$

Composition of substitutions, denoted with $\circ$, is defined as

$$t(\theta \circ \rho) = (t\theta)\rho.$$

The definition of substitution can be lifted to mapping from terms to terms as follows: Given a substitution θ and a term t, the result of applying θ to t, written $t\theta$, is defined inductively on the structure of t as follows:

1. $x\theta = \theta(x)$
2. $(s\ t)\theta = (s\theta)\ (t\theta)$
3. $(\lambda x.t)\theta = \lambda x.(t\theta)$ if x is not free in $dom(\theta)$ and $ran(\theta)$.
4. $(\lambda x.t)\theta = \lambda y.(t[x \mapsto y]\theta)$, if x is free in $dom(\theta)$ or $ran(\theta)$, and y is a new variable not free in both $dom(\theta)$ and $ran(\theta)$.

Lambda abstraction is a concise way of writing functions. This functional aspect is achieved through the *conversion rules* for λ-terms.

$$\begin{array}{ll} \alpha\text{-rule} & (\lambda x.s) \to (\lambda y.s[y/x]),\ y \notin FV(s) \\ \beta\text{-rule} & (\lambda x.s)\ t \to s[t/x] \\ \eta\text{-rule} & (\lambda x.s\ x) \to s, x \text{ not free in } s \end{array}$$

An alternative presentation of the reduction rules assumes that terms are equivalent modulo renaming of bound variables, hence the α-rule becomes redundant. We shall adopt this simplifying assumption.

We say a term t is in *β-normal form* if the β-rule does not apply to the term, i.e., t contains no β-redices. The *$\beta\eta$-normal form* is defined analogously. One of the fundamental properties of λ-calculus is the uniqueness of its normal forms, when they exist. Not every term has a normal form, for example, the term

$$(\lambda x.x\ x)(\lambda x.x\ x)$$

reduces (via the β-rule) to itself. Types can be introduced to guarantee that normal form exists for every term. The simplest type system is Church's simple type system. In this type system, we annotate every variable, free or bound, with a type. The type expressions are generated according to the following grammar:

$$\alpha, \beta ::= a \mid \alpha \to \beta.$$

That is, a type can be a *base type* (the type scheme a in the above grammar) or an *arrow type* $\alpha \to \beta$, where α and β are types. The language of terms needs to be modified slightly: λ-abstracted terms now carry the type information, e.g., as in $\lambda x_\tau.t$. Here τ is the type of the bound variable x. Types for free variables are given in a *typing environments*, which is a set of pairs of variables and types. Elements of a typing environment are usually written $x : \tau$, e.g., as in $\{x_1 : \alpha_1, \ldots, x_n : \alpha_n\}$. A *typing judgment* $\Gamma \vdash t : \tau$: asserts that the term t has type τ in the typing environment Γ (which contains all the free variables of t). These judgments are derived following the typing rules below.

$$\frac{}{\Gamma, x : \tau \vdash x : \tau :} \qquad \frac{\Gamma, x : \alpha \vdash t : \beta :}{\Gamma \vdash \lambda x_\alpha.t : \alpha \to \beta :}\ x \text{ not free in } \Gamma$$

$$\frac{\Gamma \vdash s : \alpha \to \beta \quad \Gamma \vdash t : \beta}{\Gamma \vdash (s\ t) : \beta}$$

Notational convention To avoid excessive use of parantheses, we adopt the following conventions. Applications are considered to be left-associative, e.g., the term $(((s\ x)\ y)\ z)$ is written simply as $s\ x\ y\ z$. The arrow type constructor is assumed to be right-associative, e.g., we write $\alpha \to \beta \to \gamma$ for $\alpha \to (\beta \to \gamma)$.

Formulas as simply typed terms We can represent the syntax of logical formulas as simply typed λ-terms. Logical connectives are then constants of certain types. We reserve the symbol o to denote the type of propositions, following Church. The type for unary connectives is $o \to o$, the type for binary connectives is $o \to o \to o$. Quantifiers take an argument of type $\iota \to o$, i.e., they are of type $(\iota \to o) \to o$. The types of some logical connectives are listed below:

$$\begin{array}{ll} \bot : o & \top : o \\ \vee : o \to o \to o & \wedge : o \to o \to o \\ \exists_\tau : (\tau \to o) \to o & \forall_\tau : (\tau \to o) \to o \end{array}$$

For example, a formula like $\forall x.p(x) \wedge q(x)$ is really the λ-term

$$\forall_\iota\ (\lambda x_\iota.\wedge\ \ (p\ x)\ (q\ x)).$$

We shall often use the more traditional notations when presenting inference rules. A *signature* is a set of pairs of constants and their types. An *intuitionistic sequent* is a syntactic expression of the form $\Sigma : \Delta \longrightarrow C$, where Σ is a signature, Δ is a multiset of formulas and C is a formula. We assume that all the formulas in the sequents are well-typed, given the typing context Σ.

The sequent rules for intuitionistic system is given in Figure 1. We refer to this system as IL. Cut elimination holds for IL. We denote with $\Sigma; \Gamma \vdash_{IL} C$ the fact that the sequent $\Sigma : \Gamma \longrightarrow C$ is provable using the inference rules of IL.

Theorem 1 *If there is a proof of a sequent in IL, then there is a cut-free proof of the same sequent.*

3 Abstract logic programming

The key idea to generalizing the execution mechanism behind logic programming is the interpretation of computation as *goal-directed proof search*. Logical connectives play the role of computation instructions. For example, in Prolog, when proving a goal $A \wedge B$, the interpreter tries proving both A and B. Similarly, to prove $A \vee B$, the interpreter tries proving A or B. The main challenge in this interpretation is to show that goal-directed proof search is *complete*, i.e., all theorems of the underlying logic can be proved using goal-directed proof search.

In the interpretation of computation as proof search, sequents are used to represent states of computation. In the case of logic programming, a sequent $\Sigma : \mathcal{P} \longrightarrow G$ denotes a state of an idealized interpreter. Here $\mathcal{P}$ is a multiset of formulas denoting a *logic program* and G is the goal we would like to prove from program $\mathcal{P}$. Goal directed proof search essentially means that our idealized interpreter will apply the right-introduction rules to the goal until no right-introduction rules are applicable. Note that in general, $\mathcal{P}$ cannot be arbitrary formulas if we wish the goal-directed search to be complete. For example, the intuitionistic sequent $p \vee q \longrightarrow q \vee p$ is provable, but there is no goal-directed proof of this, since neither $p \vee q \longrightarrow q$ nor $p \vee q \longrightarrow p$ is provable.

Initial and cut rules:

$$\frac{}{\Sigma : B \longrightarrow B}\ id \qquad \frac{\Sigma : \Gamma \longrightarrow B \quad \Sigma : B, \Delta \longrightarrow C}{\Sigma : \Gamma, \Delta \longrightarrow C}\ cut$$

Structural rules:

$$\frac{\Sigma : \Gamma, B, B \longrightarrow C}{\Sigma : \Gamma, B \longrightarrow C}\ contr_L \qquad \frac{\Sigma : \Gamma \longrightarrow C}{\Sigma : \Gamma, B \longrightarrow C}\ weak_L$$

Introduction rules:

$$\frac{}{\Sigma : \Gamma, \bot \longrightarrow C}\ \bot_L \qquad \frac{}{\Sigma : \Gamma \longrightarrow \top}\ \top_R$$

$$\frac{\Sigma : \Gamma, B_i \longrightarrow C}{\Sigma : \Gamma, B_1 \wedge B_2 \longrightarrow C}\ \wedge_L \qquad \frac{\Sigma : \Gamma \longrightarrow B \quad \Sigma : \Gamma \longrightarrow C}{\Sigma : \Gamma \longrightarrow B \wedge C}\ \wedge_R$$

$$\frac{\Sigma : \Gamma, B \longrightarrow C \quad \Sigma : \Gamma, D \longrightarrow C}{\Sigma : \Gamma, B \vee D \longrightarrow C}\ \vee_L \qquad \frac{\Sigma : \Gamma \longrightarrow B_i}{\Sigma : \Gamma \longrightarrow B_1 \vee B_2}\ \vee_R$$

$$\frac{\Sigma : \Gamma \longrightarrow B \quad \Sigma : \Gamma, D \longrightarrow C}{\Sigma : \Gamma, B \supset D \longrightarrow C}\ \supset_L \qquad \frac{\Sigma : \Gamma, B \longrightarrow C}{\Sigma : \Gamma \longrightarrow B \supset C}\ \supset_R$$

$$\frac{\Sigma \vdash t : \tau \quad \Sigma : \Gamma, B[t/x] \longrightarrow C}{\Sigma : \Gamma, \forall_\tau x.B \longrightarrow C}\ \forall_L \qquad \frac{y : \tau, \Sigma : \Gamma \longrightarrow B[y/x]}{\Sigma : \Gamma \longrightarrow \forall_\tau x.B}\ \forall_R, y \notin \Sigma$$

$$\frac{y : \tau, \Sigma : \Gamma, B[y/x] \longrightarrow C}{\Sigma : \Gamma, \exists_\tau x.B \longrightarrow C}\ \exists_L, y \notin \Sigma \qquad \frac{\Sigma \vdash t : \tau \quad \Sigma : \Gamma \longrightarrow B[t/x]}{\Sigma : \Gamma \longrightarrow \exists_\tau x.B}\ \exists_R$$

Figure 1: Sequent calculus for intuitionistic logic.

Definition 2 *Uniform provability.* A cut-free intuitionistic proof is a *uniform proof* if every sequent in the proof with a non-atomic succedent is the conclusion of a right-introduction rule.

Definition 3 Let $\vdash$ be a provability relation in some logic. Let $\mathcal{D}$ be a set of formulas denoting program clauses and let $\mathcal{G}$ be a set of formulas denoting goal formulas in an intended logic programming. The triple $\langle \mathcal{D}, \mathcal{G}, \vdash \rangle$ is an *abstract logic programming language* if and only if for every finite subset $\mathcal{P}$ of $\mathcal{D}$ and for every $G \in \mathcal{G}$, $\Sigma; \mathcal{P} \vdash G$ if and only if $\Sigma : \mathcal{P} \longrightarrow G$ has a uniform proof.

There is still one thing missing in our description of the interpreter for an abstract logic programming language above. That is, how should proof search proceed when the goal formula is atomic? When is the logic program $\mathcal{P}$ used in the proof search? In the following sections we shall see that this is handled through the *backchaining rules*.

4 First-order Horn clauses

In the following definition we use A to denote atomic formulas, G to denote goal formulas and D to denote program formulas. Horn clauses are defined using the following grammar.

$$G ::= \top \mid A \mid G \wedge G \mid G \vee G \mid \exists_\tau x.G$$
$$D ::= A \mid G \supset A \mid D \wedge D \mid \forall_\tau x.D$$

$$\frac{\Sigma : \mathcal{P} \xrightarrow{D} A}{\Sigma : \mathcal{P} \longrightarrow A}\ \mathit{decide} \qquad \frac{}{\Sigma : \mathcal{P} \xrightarrow{A} A}\ \mathit{initial}$$

$$\frac{\Sigma : \mathcal{P} \xrightarrow{D_1} A}{\Sigma : \mathcal{P} \xrightarrow{D_1 \wedge D_2} A}\ \wedge_L \qquad \frac{\Sigma : \mathcal{P} \xrightarrow{D_2} A}{\Sigma : \mathcal{P} \xrightarrow{D_1 \wedge D_2} A}\ \wedge_L$$

$$\frac{\Sigma : \mathcal{P} \longrightarrow G \quad \Sigma : \mathcal{P} \xrightarrow{D} A}{\Sigma : \mathcal{P} \xrightarrow{G \supset D} A}\ \supset_L \qquad \frac{\Sigma \vdash t : \tau \quad \Sigma : \mathcal{P} \xrightarrow{D[t/x]} A}{\Sigma : \mathcal{P} \xrightarrow{\forall_\tau x.D} A}\ \forall_L$$

Figure 2: Backchaining rules

A Horn logic programming interpreter is the triple $\langle \mathcal{P}, \mathcal{G}, \vdash \rangle$, where $\mathcal{P}$ is a set of D-formulas, $\mathcal{G}$ is a set of G-formulas and $\vdash$ is the intuitionistic entailment relation (in fact, it can be shown that for this fragment, classical and intuitionistic logic coincides). Goal directed proof search is complete for this Horn fragment.

Proof search for Horn clauses proceeds with right-introduction rules as usual. For the case when the goal is atomic, we use the *backchaining rules* given in Figure 2. In the figure, we use the extended sequent $\Sigma : \mathcal{P} \xrightarrow{D} A$ to indicate that the interpreter is attempting to prove A by backhaining on the program clause D. The swicth from right-introduction rules to backchaining occurs when the goal is atomic. The interpreter decides (non-deterministically) on which program clause to backchain on.

Proof search using right-introduction rules and backchaining rules is complete for the Horn fragment of intuitionistic logic [20].

5 Hereditary Harrop formulas

Recall that sequents represent states of computation. In Horn fragment, in the proof search of $\Sigma : \mathcal{P} \longrightarrow A$, the signature Σ and programs $\mathcal{P}$ are unchanged. Thus, all the computation states are actually modelled only at the atomic level, i.e., in the term structures of the atomic formulas. We now consider a richer class of formulas for which proof search introduces more dynamics into both the signature and the programs. The first-order *hereditary Harrop* (*fohh*) fragment of intuitionistic logic is defined via the following grammar.

$$G ::= \top \mid A \mid G \wedge G \mid G \vee G \mid \exists_\tau x.G \mid D \supset G \mid \forall_\tau x.G$$
$$D ::= A \mid G \supset D \mid D \wedge D \mid \forall_\tau x.D$$

Notice that we allow implication in the goal formulas. In the goal directed proof search for a sequent $\Sigma : \mathcal{P} \longrightarrow D \supset G$, the interpreter applies the $\supset_R$ rule, resulting in the sequent $\Sigma : \mathcal{P}, D \longrightarrow G$. That is, it augments a new program clause D to the existing program $\mathcal{P}$. Hence states of computation are no longer confined to atomic goals, but also program clauses. Similarly, the goal-directed search for $\Sigma : \mathcal{P} \longrightarrow \forall_\tau x.G$ results in the sequent $y : \tau, \Sigma : \mathcal{P} \longrightarrow G[y/x]$ which enriches the signature Σ with a new constant y of type τ.

6 Higher-order extensions

We can allow higher-order quantification in both Horn clauses and hereditary Harrop formulas, but some restrictions must be made on the program clauses to make sure the resulting logic still has uniform provability. In the case of higher-order hereditary Harrop formulas, unrestricted higher-order quantification may break the uniform provability of the logic. The higher-order extension of hereditary Harrop formulas (*hohh*) is defined as follows. Let $\mathcal{H}$ be the set of $\beta\eta$-normal terms that do not contain occurrences of the logical constants $\supset$ and $\bot$. An atomic formula occurrence in a formula F is said to be *rigid* if it is of the form $p\ t_1\ \cdots\ t_n$ where p is a either (non-logical) constant of type $\tau_1 \to \cdots \to \tau_n \to o$. We use A_r to range over rigid atomic formulas in $\mathcal{H}$. The notions of goal formulas and programs as given by the following grammar.

$$G ::= \top \mid A \mid G \wedge G \mid G \vee G \mid \exists_\tau x.G \mid D \supset G \mid \forall_\tau x.G$$
$$D ::= A_r \mid G \supset A_r \mid D \wedge D \mid \forall_\tau x.D$$

Note that although positive occurrences of atomic formulas in program clauses are required to be rigid formulas from $\mathcal{H}$, we shall allow arbitrary universal quantification over predicates in goal formulas. These universal quantifications are harmless as in backward proof search they will be eventually replaced by (scoped) fresh constants (i.e., eigenvariables). However, universal quantification over predicates in program clauses are allowed only when they do not affect the rigidity of its positive atomic subformulas. For example, the formula $\forall P.p \wedge q \supset P$ is not a legitimate program clause, nor is $\forall R.p \wedge q \supset R(p \supset q)$. However, $(\forall p.q \supset p) \supset r$ is a legitimate program clause, since the p here is universally quantified in a negative position in the clause, hence a positive position in a sequent (recall that program clauses always appear in a negative position in a sequent), and will therefore be replaced by a fresh constant in proof search.

An example of modular programming in hohh Here we give an example of using *hohh* to do modular programming. Suppose we want to implement a predicate *reverse* which reverses a list. That is, *reverse* L K should hold if K is the reverse of the list L. A naive way of doing this is by recursing on L and appending the recursive calls to the first element of L. In Prolog, one would write the following program (here we use λ-calculus notation, writing terms such as $f(a, b)$ as an application $(f\ a\ b)$).

```
append nil L L.
append (X::L) K (X::M) :- append L K M.
reverse nil nil.
reverse (X::L) K :- reverse L M, append M (X::nil) K.
```

A slightly more efficient way is done using an accumulator. Let us define an intermediate predicate *rv* which takes three arguments, the second being an accumulator.

```
rv nil L L.
rv (X::L) K M :- rv L (X::K) M.
reverse L K :- rv L nil K.
```

In fohh, we can "hide" the definition of rv predicate by using implication. In hohh, the predicate rv itself can be hidden by higher-order quantification. This is achieved through the following definite formula:

$$
\begin{array}{l}
\forall L \forall K. \\
(\\
\quad \forall rv.(\\
\qquad\quad (\forall L.rv\ nil\ L\ L) \wedge \\
\qquad\quad (\forall X \forall L \forall K \forall M.rv\ L\ (X :: K)\ M \supset rv\ (X :: L)\ K\ M) \\
\qquad \supset rv\ L\ nil\ K) \\
) \supset reverse\ L\ K.
\end{array}
$$

7 Further readings

Materials on abstract logic programming presented here derive from the work of Miller et. al. [16, 20, 12, 13, 8, 14]. The lecture notes by Miller [15] give a good overview of the subject. An upcoming book on higher-order logic programming by Miller and Nadathur will soon be published and contains a more comprehensive introduction to higher-order logic programming.

As we have remarked in the introduction, there are other constructive logics that can also serve as a foundation for logic programming. In particular, there have been several logic programming languages that use Girard's *linear logic*[6] as the foundation, among others, LinLog [1], Lolli [8] (based on intuitionistic linear logic) and Forum [14].

Uniform provability is a special case of a more general notion of proof search strategy called *focussing*, originally studied by Andreoli [1] for linear logic. This has resulted in a logic programming language based on linear logic called LinLog [1]. It turns out that connectives of linear logic can be divided into two sets of connectives: the *asynchrounous* connectives (essentially, their introduction rules are invertible) and *synchronous* connectives. Focussed proofs proceed by first decompose asynchronous connectives, and then pick a synchronous connective to focus on. Focussing proof search strategy is complete for linear logic and other logics. For some of the latest developments in this area, see the work of Miller et. al. [17, 9].

Logic programming languages have also been used as specification languages for modelling various operational semantics of computation systems [11, 10, 18]. The rationale behind this is that since these specification languages are based on logic, one can exploit general properties of the underlying logic to reason about properties of the encoded computation systems. Various "meta" logics have been recently designed to reason formally about specifications written in these logic programming languages. See [10, 18, 21, 2, 3, 5] for some recent work in this area. Andrew Gacek has implemented a proof assistant for the logic described in [5], called Abella (`http://abella.cs.umn.edu`), that can be used to reason about properties of λProlog programs. The Abella website contains numerous examples of λProlog programs and formal proofs of their properties, e.g., the type preservation theorem for the functional language discussed in Lecture 5.

References

[1] J.-M. Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.

[2] D. Baelde, A. Gacek, D. Miller, G. Nadathur, and A. Tiu. The Bedwyr system for model checking over syntactic expressions. In F. Pfenning, editor, *21th Conference on Automated Deduction*, number 4603 in LNAI, pages 391–397. Springer, 2007.

[3] D. Baelde and D. Miller. Least and greatest fixed points in linear logic. In N. Dershowitz and A. Voronkov, editors, *LPAR*, volume 4790 of *Lecture Notes in Computer Science*, pages 92–106. Springer, 2007.

[4] J. Duparc and T. A. Henzinger, editors. *Computer Science Logic, 21st International Workshop, CSL 2007, 16th Annual Conference of the EACSL, Lausanne, Switzerland, September 11-15, 2007, Proceedings*, volume 4646 of *Lecture Notes in Computer Science*. Springer, 2007.

[5] A. Gacek, D. Miller, and G. Nadathur. Combining generic judgments with recursive definitions. In F. Pfenning, editor, *Proceedings of LICS 2008*. IEEE Computer Society Press, 2008.

[6] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[7] J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and Types*. Cambridge University Press, 1989.

[8] J. Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994.

[9] C. Liang and D. Miller. Focusing and polarization in intuitionistic logic. In Duparc and Henzinger [4], pages 451–465.

[10] R. McDowell and D. Miller. Reasoning with higher-order abstract syntax in a logical framework. *ACM Transactions on Computational Logic*, 3(1):80–136, January 2002.

[11] R. McDowell, D. Miller, and C. Palamidessi. Encoding transition systems in sequent calculus. *Theoretical Computer Science*, 294(3):411–437, 2003.

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

[13] D. Miller. Abstractions in logic programming. In P. Odifreddi, editor, *Logic and Computer Science*, pages 329–359. Academic Press, 1990.

[14] D. Miller. Forum: A multiple-conclusion specification language. *Theoretical Computer Science*, 165(1):201–232, Sept. 1996.

[15] D. Miller. Sequent calculus and the specification of computation. In U. Berger and H. Schwichtenberg, editors, *Computational Logic*, volume 165 of *Nato ASI Series*, pages 399 – 444. Springer, 1999.

[16] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[17] D. Miller and A. Saurin. From proofs to focused proofs: A modular proof of focalization in linear logic. In Duparc and Henzinger [4], pages 405–419.

[18] D. Miller and A. Tiu. A proof theory for generic judgments. *ACM Trans. on Computational Logic*, 6(4):749–783, Oct. 2005.

[19] G. Nadathur and D. Miller. An Overview of λProlog. In *Fifth International Logic Programming Conference*, pages 810–827, Seattle, August 1988. MIT Press.

[20] G. Nadathur and D. Miller. Higher-order Horn clauses. *Journal of the ACM*, 37(4):777–814, October 1990.

[21] E. Pimentel and D. Miller. On the specification of sequent systems. In *Proc. LPAR 2005*, volume 3835 of *LNCS*, pages 352–366. Springer, 2005.