Higher-order quantification and proof search [⋆]

Dale Miller

Computer Science and Engineering, 220 Pond Lab
Pennsylvania State University
University Park, PA 16802-6106 USA
dale@cse.psu.edu

Abstract. Logical equivalence between logic programs that are first-order logic formulas holds between few logic programs, partly because first-order logic does not allow auxiliary programs and data structures to be hidden. As a result of not having such abstractions, logical equivalence will force these auxiliaries to be present in any equivalence program. Higher-order quantification can be use to hide predicates and function symbols. If such higher-order quantification is restricted so that operationally, only hiding is specified, then the cost of such higher-order quantifiers within proof search can be small: one only needs to deal with adding new eigenvariables and clauses involving such eigenvariables. On the other hand, the specification of hiding via quantification can allow for novel and interesting proofs of logical equivalence between programs. This paper will present several example of how reasoning directly on a logic program can benefit significantly if higher-order quantification is used to provide abstractions.

1 Introduction

One of the many goals of declarative programming, and particularly, logic programming, should be that the very artifact that is a program should be a flexible object about which one can reason richly. Examples of such reasoning might be partial and total correctness, various kinds of static analysis, and program transformation.

One natural question to ask in the logic programming setting is, given two logic programs, $\mathcal{P}_1$ and $\mathcal{P}_2$, written as logical formulas, is it the case that $\mathcal{P}_1$ entails $\mathcal{P}_2$ and vice versa, the notation for which we will write as $\mathcal{P}_1 \dashv\vdash \mathcal{P}_2$. In other words, are these two programs logically equivalent formulas. If this properties holds of two programs, then it is immediate that they prove the same goals: for example, if $\mathcal{P}_1 \vdash G$ and $\mathcal{P}_2 \vdash \mathcal{P}_1$ then $\mathcal{P}_2 \vdash G$. If provability of a goal from a program is described via cut-free proofs, as is often the case for logic programming [MNPS91], then cut-elimination for the underlying logic is needed to support this most basic of inferences.

[⋆] This work was supported in part by NSF grants CCR-9912387, CCR-9803971, INT-9815645, and INT-9815731 and a one month guest professorship at L'Institut de Mathématiques de Luminy, University Aix-Marseille 2 in February 2002.

But how useful is the entailment relation $\mathcal{P}_1 \dashv\vdash \mathcal{P}_2$? The answer to this depends a great deal on the logic in which these logic programs are situated. If, for example, $\mathcal{P}_1$ and $\mathcal{P}_2$ are first-order Horn clause programs (that is, conjunctions of universally quantified clauses), and entailment is taken as that for first-order classical logic, then this relationship holds for very few pairs of programs $\mathcal{P}_1$ and $\mathcal{P}_2$. The main reasons that this classical, first-order logic equivalence is nearly trivial and hence nearly worthless are outlined in the next three subsections.

1.1 Entailment generally needs induction.

Most interesting equivalences of programs will almost certainly require induction. Logics and type systems that contain induction are vital for reasoning about programs and such logics have been studied and developed to a great extent. In this paper, however, we will not be concerned with induction so that we can focus on two other important limiting aspects of logical entailment.

1.2 Classical logic is poor at encoding dynamics

Classical logic does not directly support computational dynamics, at least, not directly at the logic level. Truth in classical logic is forever; it is immutable. Computations are, however, dynamic. To code computation in classical logic, all the dynamics of a computation must be encoded as terms that are given as arguments to predicates. That is, the dynamics of computations are buried in non-logical contexts (i.e., with in atomic formulas). As a result, the dynamic aspects of computation are out of reach of logical techniques, such as modus ponens and cut-elimination.

This situation improves a bit if intuitionistic logic is used instead: the notion of truth in Kripke models involving possible worlds allows for some richer modeling of dynamics, enough, for example, to provide direct for scoping of modules and abstract datatypes [Mil89b,Mil89a]. If one selects a substructural logic, such as linear logic, for encoding logic programs, then greater encoding of computational dynamics is possible. In linear logic, for example, it is possible to model a switch that is now on but later off and to assign and update imperative programming like variables [HM94] as well as model concurrent processes that evolve independently or synchronize among themselves [AP91,Mil96]. Here, counters, imperative variables, and processes are all represented as formulas and not as terms within the scope of a predicate (non-logical constant). As a result of capturing more of the computational dynamics happening at the level of loci, logical equivalence will be richer and less trivial in establishing equivalence. Of course, for these examples, this equivalence will now be based on linear logic. All the examples that we shall present in this paper will be based on linear logic programming.

1.3 Needed abstractions are not present in first-order logics

First-order logics do not provide means to hide or abstract parts of a specification of a computation. For example, if one specifies two different specifications for sorting in first-order Horn clauses, each of with different auxiliary predicates and data structures, all of these are "visible" in first-order logic, and logical equivalence of two such programs will insist that all predicates, even auxiliary ones, must be equivalent. Logic does provide means of abstracting or hiding parts of specification and even entire data structures. This mechanism is quantification, and to hide predicates and data structures, higher-order quantification is needed. This paper will focus on illustrating how such quantification can be used to specify computation and to help in reasoning about program equivalence.

2 Quantification at function and predicate types

In the first-order logic settings, substitutions for quantifiers can determined within proof search generally using unification. Depending on what kind of higher-order quantification is allowed, it may or may not be true that unification, even higher-order unification, can determine unification entirely.

A first-order logic allows quantification over only the syntactic category individuals. Higher-order logics generally allow for quantification, also, of function symbols (quantification at function type) or predicate symbols (quantification at predicate type) or both.

For the sake of being concrete, we shall assume that the logics considered here are based on a simple type discipline, and that the type of logical formulas is o (following Church's Simple Theory of Types [Chu40]). Given this typing, if a quantifier binds a variable of type $\tau_1 \rightarrow \ldots \tau_n \rightarrow \tau_0$, where τ_0 is a primitive type, then that quantifier binds a variable at predicate type if τ_0 is o and binds a variable at function type otherwise.

Logic programming languages that allow for quantification over function symbols and enrich the term language with λ-terms provide support for the *higher-order abstract syntax* approach to representing syntactic expressions involving binders and scope [PE88]. As is familiar with, say, implementations of λProlog, Isabelle, and Elf, higher-order unification is adequate for discovering substitutions for quantifiers during proof search in languages containing quantification of individual and function types. A great deal of energy has gone into the effective implementation of such systems, including treatments of higher-order unification, explicit substitution, and search (see, for example, [NM99]). Higher-order unification [Hue75] is rather complicated, but much of higher-order abstract syntax can be maintained using a much weaker notion of unification [Mil91].

Quantification at predicate type is, however, a more complex problem. As is well known, cut-free proofs involving formulas with predicate quantification do not necessarily have the sub-formula property: sometimes predicate expressions (denoting sets and relations) are required that are not simply rearrangements of subformulas present in the sequent one is attempting to prove. Higher-order unification directly applied does not generate enough substitutions to yield completeness. Just to convince ourselves that computing predicate substitutions must be genuinely hard, imagine stating the partial correctness of a simple imperative program written using, say, assignment and interaction. Using Hoare logic for such a programming language, the correctness of a looping program is easily written by allowing predicate quantification to represent the quantification of an invariant for the loop. It is, indeed, hard to image an mechanism that would be complete for computing invariants of looping programs.

There has been some interesting work on attempts to find occasions when higher-order substitutions can be automated. See, for example, the work on set variable [Ble79,Fel00,Dow93]. The logic programming language λProlog allows some uses of higher-order quantification (used for higher-order programming), but such uses are restricted so that computing necessary predicate substitutions can be done using (essentially) higher-order (pre) unification [NM90]. This can only be done, however, for rather serious restrictions on the use of higher-order predicate substitutions (such restrictions do, however, also have a natural operational meaning within the logic programming setting).

$$\frac{\Delta \longrightarrow G[y/x]}{\Delta \longrightarrow \forall x_\tau . G}\ \forall R \qquad \frac{\Delta, D[t/x] \longrightarrow G}{\Delta, \forall x_\tau . D \longrightarrow G}\ \forall L$$

$$\frac{\Delta \longrightarrow G[t/x]}{\Delta \longrightarrow \exists x_\tau . G}\ \exists R \qquad \frac{\Delta, D[y/x] \longrightarrow G}{\Delta, \exists x_\tau . D \longrightarrow G}\ \exists L$$

Fig. 1. Inference rules for quantifiers. In both the $\exists R$ and $\forall L$ rules, t is a term of type τ. In the $\exists L$ and $\forall R$ rules, y is a variable of type τ that is not free in the lower sequent of these rules.

Figure 1 presents the sequent calculus rules for the universal and existential quantifiers. Notice that the substitution term t can be a λ-term (of type τ) and in the case that that quantification is of predicate type, the term t may contain logical connectives. Hence, as a result, the formulas $G[t/x]$ and $D[t/x]$ might have many more logical connectives and quantifiers than the formulas $\forall x.D$ and $\exists x.G$. Of course, if a sequent calculus proof does not contain the $\forall L$ and $\exists R$ inference rules at predicate type, then the treatment of higher-order quantifiers are actually quite simple: the only other possible introduction rules are $\exists$L and $\forall$R and they are simply instantiated (reading a proof bottom up) by a new "eigenvariable" y. The Teyjus implementation [NM99] of λProlog, for example, gives these a direct and effective implementation of such eigenvariable generation during proof search. It is possible to define rich collections of higher-order formulas for which one can guarantee that only predicate quantification only occurs with $\exists$L and $\forall$R. Most of the example of logic programs that we present below have this structure.

Notice that if $\mathcal{P}$ is such that all universal quantifiers of predicate type occur negatively and all existential quantifiers of predicate type occur positively, then it is easy to show that there are no occurrences of $\forall L$ or $\exists R$ in a cut-free proof of the sequent $\mathcal{P} \longrightarrow A$ where A is, say, an atomic formula. If, however, $\mathcal{P}_1$ and $\mathcal{P}_2$ are two such programs with the above restriction, there can be cut-free proofs of the sequent $\mathcal{P}_1 \longrightarrow \mathcal{P}_2$ that contain occurrences of $\forall L$ and $\exists R$.

Such sequents will generally require some substitutions that might be difficult to produce by simple automation. This is consistent with the expectation that inferences between programs require some genuine insights to establish.

We now proceed to present some example of reasoning with higher-order quantification. All of our examples involve some use of linear logic as well as higher-order quantification. We will not attempt to describe the basics of linear logic, but rather refer the reader to, say, [Gir87,Tro92]. Our first two examples will make use of linear implication, written as $\multimap$ (as as the converse $\multimapinv$) and intuitionistic implication, written as $\Rightarrow$. The multiplicative conjunction $\otimes$ appears as well. Our last example, starting in Section 5 will also make use of the multiplicative disjunction $⅋$.

As in most examples in this proposal, we shall assume the familiar Prolog convention of writing top-level implications in specification clauses in their reverse direction. Also, capital letters used as free variables will be considered universally quantified at the top level of the clause in which they appear.

3 Reversing a list is symmetric

While much of the motivation for designing logic programming languages based on linear logic has been to add expressiveness to such languages, linear logic can also help shed some light on conventional programs. In this section we consider the linear logic specification for the reverse of lists and formally show, by direct reasoning on the specification, that it is a symmetric relation [Mil].

Let the constants *nil* and $(\cdot :: \cdot)$ denote the two constructors for lists. To compute the reverse of two lists, make a place for two piles on a table. Initialize one pile to the list you wish to reverse and initialize the other pile to be empty. Next, repeatedly move the top element from the first pile to the top of the second pile. When the first pile is empty, the second pile is the reverse of the original list. For example, the following is a trace of such a computation.

$(a :: b :: c :: nil)$	nil
$(b :: c :: nil)$	$(a :: nil)$
$(c :: nil)$	$(b :: a :: nil)$
nil	$(c :: b :: a :: nil)$

In more general terms: if we wish to reverse the list L to get K, first pick a binary relation *rv* to denote the pairing of lists above (this predicate will not denote the reverse); then start with the atom $(rv\ L\ nil)$ and do a series of backchaining over the clause

$$rv\ P\ (X :: Q) \multimap rv\ (X :: P)\ Q$$

to get to the formula $(rv\ nil\ K)$. Once this is done, K is the result of reversing L. That is, if from the two formula

$$\forall P \forall X \forall Q (rv\ P\ (X :: Q) \multimap rv\ (X :: P)\ Q)$$
$$(rv\ nil\ K)$$

one can prove ($rv\ L\ nil$), then the reversing of L is K. This specification is not finished for several reasons. First, L and K are specific lists and are not quantified anywhere. Second, the relation *reverse* $L\ K$ must be linked to this sub-computation using the auxiliary predicate *rv*. Third, we can observe that of the two clauses for *rv* above, the first clause (the recursive one) can be used an arbitrary number of times during a computation while the second clause (the base case) can be used exactly once. Since we are using elements of higher-order linear logic here, linking the *rv* sub-computation to *reverse* $L\ K$ can be done using nested implications, the auxiliary predicate *rv* can be hidden using a higher-order quantifier, and the distinction between the use pattern for the inductive and base case clauses can be specified using different implications.

The entire specification of reverse can be written as the following single formula.

$$\begin{array}{r}\forall L \forall K[\ \forall rv\ (\ (\forall X \forall P \forall Q(rv\ P\ (X :: Q) \multimap rv\ (X :: P)\ Q)) \Rightarrow \\ rv\ nil\ K \multimap rv\ L\ nil) \multimap reverse\ L\ K\ ]\end{array}$$

Notice that the clause used for repeatedly moving the top elements of lists is to the left of an intuitionistic implication (so it can be used any number of times) while the formula ($rv\ nil\ K$), the base case of the recursion, is to the left of a linear implication (must be used once).

Now consider proving that reverse is symmetric: that is, if ($reverse\ L\ K$) is proved from the above clause, then so is ($reverse\ K\ L$). The informal proof of this is simple: in the table tracing the computation above, flip the rows and the columns. What is left is a correct computation of reversing again, but the start and final lists have exchanged roles. This informal proof is easily made formal by exploiting the meta-theory of higher-order quantification and of linear logic. A more formal proof proceeds as follows. Assume that ($reverse\ L\ K$) can be proved. There is only one way to prove this (backchaining on the above clause for *reverse*). Thus the formula

$$\forall rv((\forall X \forall P \forall Q(rv\ P\ (X :: Q) \multimap rv\ (X :: P)\ Q)) \Rightarrow rv\ nil\ K \multimap rv\ L\ nil)$$

is provable. Since this universally quantified expression is provable, any instance of it is also provable. Thus, instantiate it with the λ-expression $\lambda x \lambda y (rv\ y\ x)^{\perp}$ (the swapping of the arguments is one of the flips of the informal proof and the negation will perform the other flip). The resulting formula

$$(\forall X \forall P \forall Q(rv\ (X :: Q)\ P)^{\perp} \multimap (rv\ Q\ (X :: P)^{\perp})) \Rightarrow (rv\ K\ nil)^{\perp} \multimap (rv\ nil\ L)^{\perp}$$

can be simplified by using the contrapositive rule for negation and linear implication, and hence yields

$$(\forall X \forall P \forall Q(rv\ Q\ (X :: P) \multimap rv\ (X :: Q)\ P) \Rightarrow rv\ nil\ L \multimap rv\ K\ nil)$$

If we now universally generalize on *rv* we again have proved the body of the reverse clause, but this time with L and K switched.

This proof exploits the explicit hiding of the auxiliary predicate *rv* by providing a site into which a "re-implementation" of the predicate can be placed. Also notice that this proof does not have an explicit reference to induction. It is unlikely to suspect that there are many proofs of interesting properties involving list manipulation predicates that do not require induction. This example simply illustrates an aspect of higher-order quantification and linear logic in reasoning direction with specifications.

4 Two implementations of a counter

For another example of how higher-order quantification can be used to form abstractions and to enhance reasoning about code, consider the two different specifications E_1 and E_2 of a simple counter object in Figure 2 [Mil96]. Each of these specifications specify a counter using notions similar to the encapsulation of state (via linear logic) that responds to two "methods", namely, *get* and *inc*, for getting and incrementing the encapsulated value. Notice that in both of these specifications, the state is the linear atomic formula $(r\ n)$ (n is the integer denoting the counters value) and the two methods are specified by two clauses that are marked with a ! (that is, they can be invoked any number of times). In these both of these specifications, the predicate r is existentially quantified, thus properly encapsulating the state and methods.

$$\begin{aligned}
E_1 = \exists r[\ & (r\ 0) \otimes \\
& !\forall K \forall V (\mathit{get}\ V\ K \mathrel{\circ\!-} r\ V \otimes (r\ V \multimap K)) \otimes \\
& !\forall K \forall V (\mathit{inc}\ V\ K \mathrel{\circ\!-} r\ V \otimes (r\ (V+1) \multimap K))]
\end{aligned}$$

$$\begin{aligned}
E_2 = \exists r[\ & (r\ 0) \otimes \\
& !\forall K \forall V (\mathit{get}\ (-V)\ K \mathrel{\circ\!-} r\ V \otimes (r\ V \multimap K)) \otimes \\
& !\forall K \forall V (\mathit{inc}\ (-V)\ K \mathrel{\circ\!-} r\ V \otimes (r\ (V-1) \multimap K))]
\end{aligned}$$

Fig. 2. Two specifications of a global counter.

Viewed from a proof search point-of-view, these two specifications store the counter on the left side of the sequent as a linear logic assumption. The counter is then updated by "destructively" reading and then rewritten the atom used to store that value. The differences between these two implementations is that in the second of these implementations the *inc* method actually decrements the internal representation of the counter: to compensate for this choice the *get* method returns the negative of that internal value. The use of $\otimes$, !, and $\exists$ in Figure 2 is for convenience in displaying these abstract data types. For example, if we write E_1 as $\exists r(R_1 \otimes !\, R_2 \otimes !\, R_3)$, then using simple "curry/uncurry" equivalences in linear logic we can rewrite $E_1 \multimap G$ to the equivalent formula $\forall r(R_1 \multimap R_2 \Rightarrow R_3 \Rightarrow G)$. These specification are encoded using continuation passing style: the variable K ranges over continuations.

Although these two specifications of a global counter are different, they should be equivalent in some sense. Although there are several ways that the equivalence of such counters can be proved (for example, trace equivalence), the specifications of these counters are, in fact, *logically* equivalent. In particular, the entailments $E_1 \vdash E_2$ and $E_2 \vdash E_1$ are provable in linear logic. The proof of each of these entailments proceeds (in a bottom-up fashion) by choosing an eigen-variable, say s, to instantiate the existential quantifier, on the left-hand specification and then instantiating the right-hand existential quantifier with some term involving s. In both cases, it turns out that the proper term with which to instantiate that right-hand quantifier is $\lambda x.s\ (-x)$. The proof of these entailments must also use the equations

$$\{-0 = 0, -(x+1) = -x-1, -(x-1) = -x+1\}.$$

Clearly, logical equivalence is a strong equivalence: it immediately implies no logical context can tell the difference between these two implementations.

5 Multiset rewriting in proof search

To provide some more examples of direct reasoning one logical specification, we will first describe how certain aspects of security protocols can be specified in linear logic. To discuss specifications of security protocols, we first introduce multiset rewriting and how that can be encoded in proof search.

To model multiset rewriting we shall use a subset of linear logic similar to the *process clauses* introduced by the author in [Mil93]. Such clauses are simply described as follows: Let G and H be formulas composed of $\bot$, $⅋$, and $\forall$. (Think of the $⅋$ connective as the multiset constructor and $\bot$ as the empty multiset.) Process clauses are closed formulas of the form $\forall\bar{x}[G \multimap H]$ where H is not $\bot$ and all free variables of G are free in H. These clause have been used in [Mil93] to encode a calculus similar to the π-calculus. A nearly identical subset of linear logic has also been proposed by Kanovich [Kan92,Kan94]: if you write process clauses in their contrapositive form (replacing the connectives $⅋$, $\forall$, $\bot$, and $\multimapinv$ with $\otimes$, $\exists$, $\mathbf{1}$, and $\multimap$, respectively) you have what Kanovich called *linear Horn clauses.*

The multiset rewriting rule $a, b \Rightarrow c, d, e$ can be implemented as a backchaining step over the clause $c ⅋ d ⅋ e \multimap a ⅋ a ⅋ b$. That is, backchaining using this linear logic clause can be used to justify the inference rule

$$\frac{\Psi; \Delta \longrightarrow c, d, e, \Gamma}{\Psi; \Delta \longrightarrow a, a, b, \Gamma}\ ,$$

with the proviso that $c ⅋ d ⅋ e \multimap a ⅋ b$ is a member of Ψ. We can interpret this fragment of a proof as a rewriting of the multiset a, b, Γ to the multiset c, d, e, Γ by backchaining on the clause displayed above. Using the Forum presentation of linear logic [Mil96], this inference rule can be justified with the following proof fragment.

$$\dfrac{\dfrac{\dfrac{\dfrac{\Psi;\Delta \longrightarrow c,d,e,\Gamma}{\Psi;\Delta \longrightarrow c,d \parr e,\Gamma}}{\Psi;\Delta \longrightarrow c \parr d \parr e,\Gamma} \qquad \dfrac{\dfrac{}{\Psi;\cdot \xrightarrow{a} a} \quad \dfrac{}{\Psi;\cdot \xrightarrow{b} b}}{\Psi;\cdot \xrightarrow{a \parr b} a,b}}{\Psi;\Delta \xrightarrow{c \parr d \parr e \multimap a \parr b} a,b,\Gamma}}{\Psi;\Delta \longrightarrow a,b,\Gamma}$$

The sub-proofs on the right are responsible for deleting from the right-hand context one occurrence each of the atoms a and b while the subproof on the left is responsible for inserting one occurrence each of the atoms c, d, and e. Thus, we can interpret this proof fragment as a reduction of the multiset a, b, Γ to the multiset c, d, e, Γ by backchaining on the clause displayed above.

Example 1. Consider the problem of Alice wishing to communicate a value to Bob. The clause

$$\forall x[(a\ x) \parr b \multimapinv a' \parr (b'\ x)]$$

illustrates how one might synchronize Alice's agent $a\ x$ with Bob's agent b. In one, atomic step, the synchronization occurs and the value x is transfer from Alice, resulting in her continuation a', to Bob, resulting in his continuation $b'\ x$. If a server is also involved, one can imagine the clause being written as

$$\forall x[(a\ x) \parr b \parr s \multimapinv a' \parr (b'\ x) \parr s],$$

assuming that the server's state is unchanged through this interaction.

As this example illustrates, synchronization between agents is easy to specify and can trivialize both the nature of communication and the need for security protocols entirely. (For example, if such secure communications is done atomically, there is no need for a server s in the above clause.) While the clause in this example might *specify* a desired communication, it cannot be understood as an actual implementation in a distributed setting. In distributed settings, synchronization actually only takes place between agents and networks. Our main use of multiset rewriting will involve more restricted clauses with weaker assumptions about communications: these will involve synchronizations between agents and network messages (a model for asynchronous communications) and not between agents and other agents (a model of synchronous communications).

In general, however, the body of clauses are allowed to have universal quantification: since $(\forall x.Px) \parr Q$ is linear logically equivalent to $\forall x(Px \parr Q)$, we can assume such bodies are in prenex normal form (provided that x is not free in Q). Backchaining over the clause

$$\forall x_1 \ldots \forall x_i[a_1 \parr \cdots \parr a_m \multimapinv \forall y_1 \ldots \forall y_j[b_1 \parr \cdots \parr b_n]]$$

can be interpreted as multiset rewriting but where the variables $y_1, \ldots, y_j$ are instantiated with eigenvariables of the proof.

6 Security protocol in proof search

We shall briefly outline how multiset rewriting framework proposed by Cervesato, *et. al.* in MSR [CDL$^+$99,CDL$^+$00]. As we have seen, universal quantification is used can be used to handle schema variables (such as the x variable in Example 1) as well as eigenvariables: when the latter are treated properly in a sequent calculus setting, the proviso on their newness can be used to model the notions of freshness and newness in security protocols for nonces, session keys, and encryption keys.

For the sake of concreteness and simplicity, we will model messages on a network simply as terms of type *data*. We shall use a tupling operator $\langle \cdot, \cdot \rangle$ that has type $data \to data \to data$ to form composite data objects and the *unit*, written as the empty tuple $\langle\rangle$ will have the type *data*. Expressions such as $\langle \cdot, \cdot, \ldots, \cdot \rangle$ will be assumed to be built from pairing, associated to the right.

As suggested in the previous section, not just any clause makes sense in a security protocol. Various restrictions on the occurrence of predicates within clauses must be made. For example, we need to avoid that agents synchronize directly with other agents and we must avoid that one agent becomes another agent. (In Section 8 we show a different syntax for protocol clauses which will not need these various restrictions.) In general, the specification of an action possible by Alice is given by (the universal closure of) a clause of the form

$$a\ S \parr [[M_1]] \parr \cdots \parr [[M_p]] \multimapinv \forall n_1 \ldots \forall n_i [a'\ S' \parr [[M_1']] \parr \cdots \parr [[M_q']]\ ],$$

where p, q, and i are non-negative integers. This clause indicates that Alice with memory S (represented by the atom $a\ S$) inputs the p network messages $[[M_1]], \ldots, [[M_p]]$ then, in a context where $n_1, \ldots, n_i$ are new symbols, becomes the continuation a' with new memory S' and with q new output messages $[[M_1']], \ldots, [[M_q']]$. Two variants of this clause can also be allow: the first is where the atom $a\ s$ is not present in the *head* of the clause (such clauses encode agent creation) and the second is where the atom $a'\ S'$ is not present in the *body* of the clause (such clauses encode agent deletion).

7 Encryption as an abstract datatype

We now illustrate our first use of higher-order quantification at a non-predicate type. In particular, we shall model encryption keys as eigenvariables but not of type *data* but of type $data \to data$. Clearly, this is a natural choice of type since it is easy to think of encryption as being being a mapping from data to data: a particular implementation of this via 64-bit string and some particular encryption algorithm is, of course, abstracted away at this point. Building data objects using higher-type eigenvariables is a standard approach in logic programming for modeling abstract datatypes [Mil89a]. In order to place this higher-type object within data, we introduce a new constructor $\cdot^\circ$ of type $(data \to data) \to data$ that explicitly coerces an encryption function into data.

Explicit quantification of encryption keys can be used to also describe succinctly the static distribution of keys within agents. Consider, for example, the following specification.

$$\begin{array}{rcl}\exists k_{as}\exists k_{bs}[\; a\ \langle M, S\rangle & \circ\!- & a\ S \parr [[k_{as}\ M]].\\ b\ T \parr [[k_{bs}\ M]] & \circ\!- & b\ \langle M, T\rangle.\\ s\ \langle\rangle \parr [[k_{as}\ P]] & \circ\!- & s\ \langle\rangle \parr [[k_{bs}\ P]].\end{array}$$

(Here as elsewhere, quantification of capital letter variables is universal with scope limited to the clause in which the variable appears.) In this example, Alice (a) communicates with Bob (b) via a server (s). To make the communications secure, Alice uses the key k_{as} while Bob uses the key k_{bs}. The server is memoryless and only takes on the role of translating messages encrypted for Alice to messages encrypted for Bob. The use of the existential quantifiers helps establish that the occurrences of keys, say, between Alice and the server and Bob and the server, are the only occurrences of those keys. Even if more principals are added to this system, these occurrences are still the only ones for these keys. Of course, as protocols are evaluated (that is, a proof is searched for), keys may extrude their scope and move freely around the network and into the memory of possible intruders. This dynamic notion of scope extrusion is similar to that found in the π-calculus [MPW92] and is modeled here in linear logic in a way similar to an encoding given in [Mil93] for an encoding of the π-calculus into linear logic.

Example 2. As an example to illustrate the possible power of logical entailment using such quantification at higher-order type for encryption keys, consider the following two clauses:

$$a \circ\!- \forall k.[[(k\ m)]] \quad \text{and} \quad a \circ\!- \forall k.[[(k\ m')]].$$

These two clauses specify that Alice can take a step that generates a new encryption key and then outputs either the message m or m' encrypted. Since Alice has no continuation, no one will be able to decode this message. It should be the case that these two clauses are equivalent, but in what sense? It is an easy matter to show that these two clauses are actually logically equivalent. A proof that the first implies the second contains a subproof of the sequent

$$\forall k.[[(k\ m')]] \longrightarrow \forall k.[[(k\ m)]],$$

and this is proved by introducing the eigenvariable, say c, on the right and the term $\lambda w.(c\ m)$ on the left.

Public key encryption can be encoded using a "key" of type $data \rightarrow data \rightarrow data$ as the following clause illustrates:

$$\begin{array}{l}\exists k.[\; a\ S \parr [[(k\ N\ M)]] \circ\!- a'\ \langle S, M\rangle.\\ \qquad pubkey\ alice\ M \circ\!- \forall n.[[(k\ n\ M)]].]\end{array}$$

We have introduced now an auxiliary predicate for storing public keys: this is reasonable since they should be something that can be looked up in a registry.

$$
\begin{array}{rcl}
\multicolumn{3}{l}{\exists k_{as} \exists k_{bs} \{} \\
a\ S & \mathrel{\circ\!-} & \forall na.\ a\ \langle na, S\rangle \parr [[\langle alice, bob, na\rangle]]. \\
a\ \langle N, S\rangle \parr [[(k_{as}\langle N, bob, K, En\rangle)]] & \mathrel{\circ\!-} & \qquad a\ \langle N, K, S\rangle \parr [[En]]. \\
a\ \langle Na, Key^{\circ}, S\rangle \parr [[(Key\ Nb]]) & \mathrel{\circ\!-} & \qquad a\ \langle\rangle \parr [[(Key\ \langle Nb, S\rangle)]]. \\
b\ \langle\rangle \parr [[(k_{bs}\ \langle Key^{\circ}, alice\rangle)]] & \mathrel{\circ\!-} & \forall nb.\ b\ \langle nb, Key^{\circ}\rangle \parr [[(Key\ nb)]]. \\
b\ \langle Nb, Key\rangle \parr [[(Key\langle Nb, S\rangle)]] & \mathrel{\circ\!-} & \qquad b\ S. \\
s \parr [[\langle alice, bob, N\rangle]] & \mathrel{\circ\!-} & \forall k.\ \ s \parr [[(k_{as}\langle N, bob, k^{\circ}, k_{bs}\langle k^{\circ}, alice\rangle\rangle)]]. \\
\multicolumn{3}{l}{\}}
\end{array}
$$

Fig. 3. Encoding the Needham-Schroeder protocol.

For example, the following clause describes a method for Bob to send Alice a message using her public key.

$$\forall M \forall S.\ b\ \langle M, S\rangle \mathrel{\circ\!-} b'\ S \parr pubkey\ alice\ M.$$

Every instance of the call to *pubkey* places a new nonce into the encrypted data and only Alice has the full key that makes it possible for her to ignore this nonce and only decode the message.

For a more interesting example, we specify the Needham-Schroeder Shared Key protocol (following [SC01]) in Figure 3. Notice that two shared keys are used in this example and that the server creates a new key that is placed within data and is then used for Alice and Bob to communicate directly. Notice also that it is easy to show that this protocol implements the specification (taken from Example 1):

$$\forall x[(a\ x) \parr b \parr s \mathrel{\circ\!-} a' \parr (b'\ x) \parr s].$$

That is, the formula displayed in Figure 3 logically entails the above displayed formula. The linear logic proof of this entailment starts with the multiset $(a\ c), b, s$ on the right of the sequent arrow (for some "secret" eigenvariable c) and then reduces this back to the multiset $a', (b'\ c), s$ simply by "executing" the logic program in Figure 3. Notice that the $\forall$ used in the bodies of clauses in this protocol are used both for nonce creation (at type *data*) and encryption key creation (at type $data \rightarrow data$).

8 Abstracting over internal states of agents

Existential quantification over program clauses can also be used to hide predicates encoding agents. In fact, one might argue that the various restrictions on sets of process clauses (no synchronization directly with atoms encoding agents, no agent changing into another agent, etc) might all be considered a way to enforce locality of predicates. Existential quantification can, however, achieve this same notion of locality, but much more declaratively.

First notice that such quantification can be used to encode 3-way synchronization using 2-way synchronization via a hidden intermediary. For example, the following entailment is easy to prove in linear logic.

$$\exists\, x. \left\{ \begin{array}{l} a \parr b \circ\!- \; x \\ x \parr c \circ\!- \; d \parr e \end{array} \right\} \quad \dashv\vdash \quad a \parr b \parr c \circ\!- \; d \parr e$$

In a similar fashion, intermediate states of an agent can be taken out entirely. For example,

$$\exists\, a_2, a_3. \left\{ \begin{array}{l} a_1 \parr [[m_0]] \circ\!- \; a_2 \parr [[m_1]] \\ a_2 \parr [[m_2]] \circ\!- \; a_3 \parr [[m_3]] \\ a_3 \parr [[m_4]] \circ\!- \; a_4 \parr [[m_5]] \end{array} \right\} \quad \dashv\vdash$$

$$a_1 \parr [[m_0]] \circ\!- \; ([[m_1]] \circ\!- \; ([[m_2]] \circ\!- \; ([[m_3]] \circ\!- \; ([[m_4]] \circ\!- \; ([[m_5]] \parr a_4)))))$$

This suggests an alternative syntax for agents.

So far, we have only considered formulas in their "bipolar form". That is, clauses of the form

$$P_1 \parr \cdots \parr P_n \multimap Q_1 \parr \cdots \parr Q_m \quad \text{or equivalently} \quad (P_1^\perp \otimes \cdots \otimes P_n^\perp) \parr Q_1 \parr \cdots \parr Q_m.$$

Such a formula is a nesting of synchronous connectives within asynchronous connectives, but no synchronous connective is in the scope of an asynchronous connective. Andreoli [And92] showed that if we allow the addition of additional constants, arbitrary formulas of linear logic can be "compiled" to a collection of bipolar forms: basically, when the nesting alternates once, simply introduce new constants and have these be defined to handle the meaning of the next alternation, and so on.

We can now see that the specifications that we have been using so far are actually the result of compiling more general clauses into bipolars. To that end, consider the following syntactic categories of linear logic formulas:

$$H ::= A \mid \perp \mid H \parr H \mid \forall x.H$$

$$D ::= H \mid D \multimap H \mid \forall x.D$$

Notice that $\mathbf{1}$ belongs to D since it is equivalent to $\perp \multimap \perp$. Also, if one skips a phase, the two phases can be contracted (if no intervening quantifiers) as follows:

$$p \circ\!- \; (\perp \circ\!- \; (q \circ\!- \; k)) \equiv p \parr q \circ\!- \; k$$

Notice that $\forall x(px \parr qx)$ is not equivalent to $\forall x(px) \parr \forall x(qx)$ (the forward direction does not hold). An encoding of the Needham-Schroeder Shared Key protocol in this style syntax is given in Figure 4.

Formulas such as these can be seen as *logically equivalent* to clauses in process theories: the gap between them is only predicate quantification. This new set of formulas might also be argued to be superior to the form using bipolars for a number of reasons. First, the restrictions that we need to make on occurrences on predicates in various clauses of a protocol specification can be removed entirely. In fact, only one predicate symbol, the one denoting network messages,

$$\begin{array}{ll}
\text{(Out)} & \forall na.[[\langle alice,\ bob,\ na\rangle]] \mathrel{\circ\!-} \\
\text{(In)} & \quad (\forall Kab \forall En.[[kas\langle na,\ bob,\ Kab^{\circ},\ En\rangle]] \mathrel{\circ\!-} \\
\text{(Out)} & \qquad ([[En]] \mathrel{\circ\!-} \\
\text{(In)} & \qquad\quad (\forall NB.[[(KabNB)]] \mathrel{\circ\!-} \\
\text{(Out)} & \qquad\qquad ([[(Kab(NB,\ secret))]])))).
\end{array}$$

$$\begin{array}{ll}
\text{(Out)} & \perp \mathrel{\circ\!-} \\
\text{(In)} & \quad (\forall Kab.[[(kbs(Kab^{\circ},\ alice))]] \mathrel{\circ\!-} \\
\text{(Out)} & \qquad (\forall nb.[[(Kabnb)]] \mathrel{\circ\!-} \\
\text{(In)} & \qquad\quad ([[(Kab(nb,\ secret))]] \mathrel{\circ\!-} \\
\text{(Cont)} & \qquad\qquad b\ secret))).
\end{array}$$

$$\begin{array}{ll}
\text{(Out)} & \perp \Leftarrow \\
\text{(In)} & \quad (\forall N.[[\langle alice,\ bob,\ N\rangle]] \mathrel{\circ\!-} \\
\text{(Out)} & \qquad (\forall key.[[kas\langle N,\ bob,\ key^{\circ},\ kbs(key^{\circ},\ alice)\rangle]])).
\end{array}$$

Fig. 4. Encodings of Alice, Bob, and the server (respectively)

is needed. Second, there is a strong parallel between this second style specification and process calculus representation of communicating agents, at least in the sense that formulas will appear to be built from input and output prefixes. Third, agents that are prepared to output and those prepared to do inputs will be duals of one another and, as such, will organize themselves within sequents nicely: formulas denoting agents willing to output will appear on the right and formulas denoting agents willing to input will appear on the left of the sequent arrow. (Remember the proof theoretic distinction in which asynchronous behavior occurs on the right and synchronous behavior occurs on the left.) Finally, we do not need to keep track of memory explicitly: memory is modeled by values and variables that occur in the "continuation" to a process formula.

9 Conclusion

We have shown than abstractions in the specification of logic programs via quantification at higher-order types can improve the chances that one can perform interesting inferences directly on the logic specification. Induction and co-induction will certainly be important, if not central, to establishing a most logical entailments involving logic specifications. Abstractions of the form we have discussed here, however, will most like play an important role in any comprehensive approach to reasoning about logic programming specifications.

References

[And92] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.

[AP91] J.M. Andreoli and R. Pareschi. Linear objects: Logical processes with built-in inheritance. *New Generation Computing*, 9(3-4):445–473, 1991.

[Ble79] W. W. Bledsoe. A maximal method for set variables in automatic theorem-proving. In *Machine Intelligence 9*, pages 53–100. John Wiley & Sons, 1979.

[CDL$^+$99] Iliano Cervesato, Nancy A. Durgin, Patrick D. Lincoln, John C. Mitchell, and Andre Scedrov. A meta-notation for protocol analysis. In R. Gorrieri, editor, *Proceedings of the 12th IEEE Computer Security Foundations Workshop — CSFW'99*, pages 55–69, Mordano, Italy, 28–30 June 1999. IEEE Computer Society Press.

[CDL$^+$00] Iliano Cervesato, Nancy A. Durgin, Patrick D. Lincoln, John C. Mitchell, and Andre Scedrov. Relating strands and multiset rewriting for security protocol analysis. In P. Syverson, editor, *13th IEEE Computer Security Foundations Workshop — CSFW'00*, pages 35–51, Cambrige, UK, 3–5 July 2000. IEEE Computer Society Press.

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

[Dow93] Gilles Dowek. A complete proof synthesis method for the cube of type systems. *Journal of Logic and Computation*, 3(3):287—315, 1993.

[Fel00] Amy Felty. The calculus of constructions as a framework for proof search with set variable instantiation. *Theoretical Computer Science*, 232(1-2):187–229, February 2000.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[HM94] Joshua Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994.

[Hue75] Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[Kan92] Max Kanovich. Horn programming in linear logic is NP-complete. In *Proceedings of the Seventh Annual IEEE Synposium on Logic in Computer Science*, pages 200–210. IEEE Computer Society Press, June 1992.

[Kan94] Max Kanovich. The complexity of Horn fragments of linear logic. *Annals of Pure and Applied Logic*, 69:195–241, 1994.

[Mil] Dale Miller. An overview of linear logic programming. To appear in a book on linear logic, edited by Thomas Ehrhard, Jean-Yves Girard, Paul Ruet, and Phil Scott. Cambridge University Press.

[Mil89a] Dale Miller. Lexical scoping as universal quantification. In *Sixth International Logic Programming Conference*, pages 268–283, Lisbon, Portugal, June 1989. MIT Press.

[Mil89b] Dale Miller. A logical analysis of modules in logic programming. *Journal of Logic Programming*, 6(1-2):79–108, January 1989.

[Mil91] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[Mil93] Dale Miller. The π-calculus as a theory in linear logic: Preliminary results. In E. Lamma and P. Mello, editors, *Proceedings of the 1992 Workshop on Extensions to Logic Programming*, number 660 in LNCS, pages 242–265. Springer-Verlag, 1993.

[Mil96] Dale Miller. Forum: A multiple-conclusion specification language. *Theoretical Computer Science*, 165(1):201–232, September 1996.

[MNPS91] 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.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, Part I. *Information and Computation*, pages 1–40, September 1992.

[NM90] Gopalan Nadathur and Dale Miller. Higher-order Horn clauses. *Journal of the ACM*, 37(4):777–814, October 1990.

[NM99] Gopalan Nadathur and Dustin J. Mitchell. System description: Teyjus—a compiler and abstract machine based implementation of Lambda Prolog. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction*, pages 287–291, Trento, Italy, July 1999. Springer-Verlag LNCS.

[PE88] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208. ACM Press, June 1988.

[SC01] Paul Syverson and Iliano Cervesato. The logic of authentication protocols. In R. Focardi and R. Gorrieri, editors, *Foundations of Security Analysis and Design*, volume LNCS 2171. Springer-Verlag, 2001.

[Tro92] Anne S. Troelstra. *Lectures on Linear Logic.* CSLI Lecture Notes 29, Center for the Study of Language and Information, Stanford, California, 1992.