Logic Programming in a Fragment of Intuitionistic Linear Logic *

Joshua S. Hodas
Computer Science Department
Harvey Mudd College
Claremont, CA 91711-5990 USA
hodas@cs.hmc.edu

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

Abstract

When logic programming is based on the proof theory of intuitionistic logic, it is natural to allow implications in goals and in the bodies of clauses. Attempting to prove a goal of the form $D \supset G$ from the context (set of formulas) Γ leads to an attempt to prove the goal G in the extended context $\Gamma \cup \{D\}$. Thus during the bottom-up search for a cut-free proof contexts, represented as the left-hand side of intuitionistic sequents, grow as stacks. While such an intuitionistic notion of context provides for elegant specifications of many computations, contexts can be made more expressive and flexible if they are based on linear logic. After presenting two equivalent formulations of a fragment of linear logic, we show that the fragment has a goal-directed interpretation, thereby partially justifying calling it a logic programming language. Logic programs based on the intuitionistic theory of hereditary Harrop formulas can be modularly embedded into this linear logic setting. Programming examples taken from theorem proving, natural language parsing, and data base programming are presented: each example requires a linear, rather than intuitionistic, notion of context to be modeled adequately. An interpreter for this logic programming language must address the problem of splitting contexts; that is, when attempting to prove a multiplicative conjunction (tensor), say $G_1 \otimes G_2$, from the context Δ, the latter must be split into disjoint contexts Δ_1 and Δ_2 for which G_1 follows from Δ_1 and G_2 follows from Δ_2. Since there is an exponential number of such splits, it is important to delay the choice of a split as much as possible. A mechanism for the lazy splitting of contexts is presented based on viewing proof search as a process that takes a context, consumes part of it, and returns the rest (to be consumed elsewhere). In addition, we use collections of Kripke interpretations indexed by a commutative monoid to provide models for this logic programming language and show that logic programs admit a canonical model.

1 Introduction

Fragments of intuitionistic first-order and higher-order logics are commonly used as specification languages and logic programming languages. For example, first-order and higher-order versions of *hereditary Harrop formulas* (formulas with no positive occurrences of disjunctions or existential quantifiers) have been used both as specification languages for proof systems (Felty, 1993; Felty and Miller, 1988; Paulson, 1990; Pfenning, 1988) and as the basis of logic programming languages (Gabbay and Reyle, 1984; Hallnäs and Schroeder-Heister, 1990; McCarty, 1988; Miller, 1990; Miller, Nadathur, Pfenning, and Scedrov, 1991). Part of the expressiveness of such systems derives from

*This paper will appear in a special issue of *Journal of Information and Computation* in mid 1994.

the proof rule that states that in order to prove an implication $D \supset G$ from the context (set of assumptions) Γ, first augment the context with D and attempt a proof of G in the new context. That is, the sequent $\Gamma \longrightarrow D \supset G$ has a proof if and only if $\Gamma \cup \{D\} \longrightarrow G$ has a proof.

The stack-like left-hand side of sequents in intuitionistic sequent proofs can be exploited by programs in many ways. In theorem provers, they can be used to store the current assumptions and eigen-variables of a proof; in natural language parsers, they can be used to store assumed gaps when parsing relative clauses; in data base programs, they can be used to store the state of the data base; in logic programs, they can be used to provide a basis for modular programming, local declarations, and abstract data types.

While intuitionistic contexts naturally address computing concerns in a large number of applications, in others they are too limiting. One problem that appears frequently is that, speaking operationally, once an item is placed into a context, it is not possible to remove it, short of stopping the process that created the context. Since the contraction rule is freely available in intuitionistic logic, contexts can always be assumed to grow as the proof is developed from the bottom up. Such monotonicity is problematic in numerous settings.

- When using an intuitionistic meta-logic to design theorem provers it is natural to use the meta-logic's context to manage object-level hypotheses and eigen-variables (Felty and Miller, 1988; Paulson, 1990). With such an approach, however, there is no logical way to specify any variations of the contraction rule for the object logic: arbitrary contraction on all hypotheses is imposed by the meta-logic.

- A proposed technique for parsing relative clauses is to first assume the existence of a noun phrase (a *gap*) and then attempt to parse a sentence (Pareschi, 1989). Intuitionistic contexts do not naturally enforce the constraint that the assumed gap must be used while parsing the relative clause and that the gap cannot appear in certain positions ("island constraints" (Pereira and Shieber, 1987)).

- Intuitionistic contexts can be used to manage a data base. While adding facts, querying facts, and performing hypothetical reasoning ("if I pass CS121, will I graduate") are easy to model using intuitionistic contexts, updating and retracting facts cannot be modeled straightforwardly (Bonner, McCarty, and Vadaparty, 1989; Gabbay and Reyle, 1984; Miller, 1989b).

- A notion of state encapsulation (as in object-oriented programming) can be approximated using intuitionistic logic (Hodas and Miller, 1990) by representing an object's state with assumptions in a context. Updating that state, however, means changing those representative assumptions, and the only change allowed with intuitionistic contexts is augmentation. Thus, as computation progresses, an object's state becomes progressively more non-deterministic: seldom the desired notion of state.

Each of these problems can be addressed by adopting a more refined notion of context. In this paper, which is a revision and extension of a paper given at the 1991 Logic in Computer Science Symposium (Hodas and Miller, 1991), we present a fragment of linear logic that makes a suitable logic programming language and permits very natural solutions to all of the above problems.

For the purposes of this paper we will characterize logic programming languages by concentrating only on logical connectives and quantifiers of first-order logic. We will not address notions of control: in particular, we will equate the "execution" of logic programs with the non-deterministic bottom-up search for certain kinds of proofs. We shall mostly ignore the large number of issues that are involved in converting specifications of computations, of the sort given here, to real computations. These issues are currently being studied by the authors.

2 Logic programming language design

Not all logics appear to be appropriate as the foundation of a logic programming language: while a weak logic such as Horn clauses clearly is appropriate for such a use, many richer logics do not seem to be. In a sense, logic programming should be based as much on a notion of "goal-directed search" as on the fact that it makes use of the syntax and semantics of logic. Full first-order logic, for example, does not support this notion of goal-directed search. In previous work goal-directed search was formalized using the concept of *uniform* sequent proof (Miller, 1989b; Miller et al., 1991). In this section we review the definition of uniform proofs and present a logic programming language based on intuitionistic (actually minimal) logic that significantly extends Horn clauses. It is this logic programming language that we shall refine with linear logic connectives in the next section.

It has been argued in various places, for example (Miller, 1989b; Miller et al., 1991), that evaluation in logic programming is the search for certain simple, cut-free, sequent proofs. In such a view, a sequent $\Gamma \longrightarrow G$ denotes the state of an interpreter that is attempting to determine whether the goal G follows from the program Γ. Goal-directed search is characterized operationally by the bottom-up construction of proofs in which right-introduction rules are applied first and left-introduction rules are applied only when the right-hand side is atomic. This is equivalent to saying that the logical connectives in a goal are decomposed uniformly and independently from the program: the program is only considered when the goal has a non-logical constant for its head — that is, when it is atomic. This idea is formalized for single conclusion sequent systems with the following definitions.

Definition 1 *A cut-free sequent proof is a* uniform proof *if for every occurrence in the proof of a sequent whose right-hand side is not atomic, that sequent is the conclusion of a right-introduction rule.*

Definition 2 *Let $\mathcal{D}$ and $\mathcal{G}$ be (possibly infinite) sets of formulas. The triple $\langle \mathcal{D}, \mathcal{G}, \vdash \rangle$ is an* (abstract) logic programming language *if for every finite subset $\Gamma \subseteq \mathcal{D}$ and for every $G \in \mathcal{G}$, the sequent $\Gamma \longrightarrow G$ has a proof in the proof system $\vdash$ if and only if it has a uniform proof in $\vdash$. The set $\mathcal{D}$ represents those formulas that are taken to be program clauses and the set $\mathcal{G}$ are those formulas that are taken to be goals.*

Clearly, full first-order classical and intuitionistic logics are not logic programming languages. That is, if $\mathcal{N}$ is taken to be all first-order formulas and $\vdash$ is taken to be either classical or intuitionistic provability, then the triple $\langle \mathcal{N}, \mathcal{N}, \vdash \rangle$ is not a logic programming language, since in each case there are provable sequents, such as $p \vee q \longrightarrow q \vee p$, that have no uniform proofs.

An intuitionistic sequent calculus $\mathcal{I}$ for the logical connectives $true, \wedge, \supset$, and $\forall$ is given in Figure 1. Here, the left-hand side of a sequent is intended to be a set: thus the structural rules of exchange and contraction are not needed. This follows from the fact that the pattern Γ, a (denoting the set union of Γ and $\{a\}$) matches the set $\{a, b, c\}$ in two ways: one assigns Γ to $\{a, b, c\}$ and the other to $\{b, c\}$. Because of the form of the *identity* inference, the structural rule for weakening is also not required. It should be noted that it is possible to formulate this logic with multisets of formulas (rather than sets), in which case the structural rules (except for exchange) would need to be made explicit. That formulation is relevant to the first formulation of linear logic in the next section.

The expression $\Gamma \vdash_{\mathcal{I}} G$ denotes the proposition that there is an $\mathcal{I}$-proof of the sequent $\Gamma \longrightarrow G$. Gentzen's proof of cut-elimination (Gentzen, 1969) can be used to show that the cut rule in Figure 2 is admissible in $\mathcal{I}$. Although it is possible to require $\Gamma = \Gamma'$ in the cut rule, the more general form

$$\frac{}{\Gamma, B \longrightarrow B}\ identity \qquad \frac{}{\Gamma \longrightarrow true}\ trueR$$

$$\frac{\Gamma, B_1, B_2 \longrightarrow C}{\Gamma, B_1 \wedge B_2 \longrightarrow C}\ \wedge L \qquad \frac{\Gamma \longrightarrow B \qquad \Gamma \longrightarrow C}{\Gamma \longrightarrow B \wedge C}\ \wedge R$$

$$\frac{\Gamma \longrightarrow B \qquad \Gamma, C \longrightarrow E}{\Gamma, B \supset C \longrightarrow E}\ \supset L \qquad \frac{\Gamma, B \longrightarrow C}{\Gamma \longrightarrow B \supset C}\ \supset R$$

$$\frac{\Gamma, B[t/x] \longrightarrow C}{\Gamma, \forall x.B \longrightarrow C}\ \forall L \qquad \frac{\Gamma \longrightarrow B[y/x]}{\Gamma \longrightarrow \forall x.B}\ \forall R,$$

provided that y is not free in the lower sequent.

Figure 1: The proof system $\mathcal{I}$ for a fragment of intuitionistic logic.

$$\frac{\Gamma' \longrightarrow B \qquad \Gamma, B \longrightarrow C}{\Gamma' \longrightarrow C}\ cut, \quad \text{provided } \Gamma \subseteq \Gamma'.$$

Figure 2: The cut-rule for $\mathcal{I}$.

given is useful in showing certain model-theoretic results. Cut will be stated in a similar form for the proof system given in Figure 7 and that form of cut will be used to advantage in Section 6 where a semantic result is presented.

Proposition 1 *The triple $\langle \mathcal{N}_0, \mathcal{N}_0, \vdash_{\mathcal{I}} \rangle$, where $\mathcal{N}_0$ is the set of all formulas built from the logical constants true, $\wedge, \supset$, and $\forall$, and where $\vdash_{\mathcal{I}}$ is intuitionistic provability, is a logic programming language.*

This proposition is proved by showing that given an $\mathcal{I}$-proof it is always possible to permute enough inference rules to make it uniform. For a closely related proof see (Miller, 1989b). The main proof in (Miller et al., 1991) is concerned with a much stronger language that includes some forms of function and predicate quantification.

It is possible to constrain uniform proofs in this logic even more and still not lose completeness. In particular, it is apparent from the proof of the last proposition that left-introduction rules are only needed to support *backchaining*. This observation involves two parts: first, backchaining is a composition of several left-introduction rules and second, when an atomic goal is to be proved, there must be some particular formula on the left that can be processed completely to provide a subproof of that atomic goal. By extending this observation, Andreoli has developed an interesting generalization of backchaining, called *focusing* (Andreoli, 1992).

These observations about backchaining are captured in the following proof system. Let B be a formula over the logical constants $true, \wedge, \supset$, and $\forall$, and define $|B|$ to be the smallest set of pairs such that

1. $\langle \emptyset, B \rangle \in |B|$,

2. if $\langle \Delta, B_1 \wedge B_2 \rangle \in |B|$ then both $\langle \Delta, B_1 \rangle \in |B|$ and $\langle \Delta, B_2 \rangle \in |B|$,

3. if $\langle \Delta, \forall x.B' \rangle \in |B|$ then for all closed terms t, $\langle \Delta, B'[t/x] \rangle \in |B|$, and

4. if $\langle \Delta, G \supset B' \rangle \in |B|$ then $\langle \Delta \cup \{G\}, B' \rangle \in |B|$.

$$\frac{\Gamma \longrightarrow G_1 \quad \ldots \quad \Gamma \longrightarrow G_n}{\Gamma \longrightarrow A}\ BC,$$

provided $n \geq 0$, A is atomic, $B \in \Gamma$, and $\langle\{G_1, \ldots, G_n\}, A\rangle \in |B|$.

Figure 3: Backchaining for $\mathcal{I}$.

Informally, if $\langle\Delta, A\rangle \in |B|$ then the formula B can be used to establish the formula A if each of the formulas in the set Δ can be established; that is, A might be proved by backchaining over B. Furthermore, backchaining can be limited to the case where the formula A is atomic. Let $\mathcal{I}'$ be the proof system that results from replacing the *identity*, $\supset L, \wedge L$, and $\forall L$ rules in Figure 1 with the *backchaining* inference rule in Figure 3.

Proposition 2 *Let $\Gamma \cup \{B\}$ be a set of formulas over true, $\wedge, \supset$, and $\forall$. Then, the sequent $\Gamma \longrightarrow B$ has a proof in $\mathcal{I}$ if and only if it has a proof in $\mathcal{I}'$.*

Again, the proof of this follows from the permutability inference rules. Note that there is only one left-rule in $\mathcal{I}'$, namely BC, and proofs in $\mathcal{I}'$ are necessarily uniform since BC applies only to sequents with atomic right-hand sides. The $\mathcal{I}'$ proof system provides a useful starting point for the implementation of an interpreter for this logic programming language.

Since it is only the impermutability of the left-hand rules for disjunction and existential quantification that keep uniform proofs from being complete for full first-order intuitionistic logic, it is possible to introduce disjunctions and existential quantifiers as long as they never need to be introduced on the left. This is possible if they have only positive occurrences in (cut-free) proofs: that is, if they appear only positively in formulas on the right of sequents and negatively in formulas on the left of sequents. There are at least two ways that such a restriction can be maintained.

First, define the sets $\mathcal{D}_0$ and $\mathcal{G}_0$ to be the D and G-formulas given by the following mutual recursion:

$$D := true \mid A \mid D_1 \wedge D_2 \mid G \supset D \mid \forall x.D$$
$$G := true \mid A \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid \exists x.G \mid D \supset G \mid \forall x.G.$$

If the $\mathcal{I}$-proof system is extended with the introduction rules for $\vee$ and $\exists$, the triple $\langle\mathcal{D}_0, \mathcal{G}_0, \vdash_{\mathcal{I}}\rangle$ is a logic programming language. The proof of this does not differ significantly from the proof of Proposition 1. It is, in fact, this language that is referred to as first-order *hereditary Harrop formulas* in (Miller et al., 1991).

Alternately, we can use a slightly higher-order variant of the logic over just *true*, $\wedge, \supset$, and $\forall$ to "define" part of the meaning of disjunctions and existential quantifiers. In particular, consider the three higher-order Horn clauses (see Nadathur and Miller, 1990, for a treatment of such clauses):

$$\forall P \forall Q[P \supset (P \vee Q)] \qquad \forall P \forall Q[Q \supset (P \vee Q)] \qquad \forall B \forall T[(B\ T) \supset \exists B]$$

Here, $\vee$ and $\exists$ are treated as non-logical symbols that have the types (as in Church's Simple Theory of Types (Church, 1940)) $o \to o \to o$ and $(i \to o) \to o$, respectively, where o is the type of propositions and i is the type of individuals. These clauses encode the right-introduction rules for disjunctions and existential quantifiers. In order to enforce the fact that these three clauses are to act as definitions, it is necessary to restrict occurrences of the non-logical constants $\vee$ and $\exists$ as in the paragraph above: $\vee$ and $\exists$ can have no negative occurrences in a goal and no positive occurrences in program clauses other than the three clauses displayed above. This ensures that the only clauses that can be used to prove a disjunctive or existential goal are those given above. These two approaches amount to a description of the same logic programming language.

Throughout this discussion, the cut-elimination theorem has not played a major role, since computation in logic programming has been identified with the search for *cut-free* proofs. As we shall show in Section 6, the cut-elimination theorem plays the important "external" role of providing canonical models for logic programming languages.

3 A linear logic programming language

It is possible to assume, without loss of generality, that $\mathcal{I}'$-proofs have the following property: if the sequents $\Gamma \longrightarrow B$ and $\Gamma' \longrightarrow B'$ have occurrences on the same path in a proof, with $\Gamma \longrightarrow B$ being closer to the endsequent, then $\Gamma \subseteq \Gamma'$. Thus, as a computation builds a proof from the bottom up, the left-hand sides of sequents do not decrease. This limitation on the sort of changes that are allowed means that intuitionistic contexts are too simplistic for many desired uses of contexts in logic programming. Since contexts can be assumed to be non-decreasing, formulas in them are available for backchaining on any number of times; they represent unbounded resources for constructing proofs. Linear logic offers a natural setting where notions of bounded and unbounded resource can be developed.

In order to refine the logic programming language described in Section 2, we consider the linear logic connectives $\top, \&, \otimes, \multimap, !$, and $\forall$. Proof rules for these connectives are given in Figure 4 and a cut rule for this proof system is given in Figure 5. Here, the left-hand side of sequents are multisets of formulas. As a result, the structural rule for exchange need not be explicitly stated. The structural rules of contraction and weakening are given as the inference rules $!C$ (for contraction) and $!W$ (for weakening), but they are only available for formulas of the form $!B$. The syntactic variable $!\Delta$ denotes the multiset $\{!C \mid C \in \Delta\}$. We write $\Delta \vdash_{LL} B$ if the sequent $\Delta \longrightarrow B$ has a proof in the proof system of Figure 4. Because all sequents in Figure 4 are single conclusion sequents, we shall be working completely within the "intuitionistic" fragment of linear logic.

It is easy to see that linear logic, even over just the logical connectives considered here, is not an abstract logic programming language. For example, the sequents $a \otimes b \longrightarrow b \otimes a$, $!a \longrightarrow !a \otimes !a$, $!a \& b \longrightarrow !a$, and $b \otimes (b \multimap !a) \longrightarrow !a$ are all provable in intuitionistic linear logic but do not have uniform LL-proofs. The problem here is that $\otimes R$ and $!R$ do not permute down over all the left-introduction rules. For this reason, we consider, instead, a fragment of linear logic that contains neither $!$ nor $\otimes$ as connectives, although it retains some of their functionality. We do this by making two changes to the formulation of linear logic given in Figure 4. First, sequents will be of the form $\Gamma; \Delta \longrightarrow B$ where B is a formula, Γ is a set of formulas, and Δ is a multiset of formulas. Such sequents have their context divided into two parts: the unbounded part, Γ, that corresponds to the left-hand side of intuitionistic sequents, and the bounded part, Δ, which corresponds to left-hand side of sequents of the purely linear fragment of linear logic (no !'s). Contraction and weakening are allowed in the unbounded part of the context, but not in the bounded part. As we show below, the sequent $B_1, \ldots, B_n; C_1, \ldots, C_m \longrightarrow B$ can be mapped to the linear logic sequent

$$!B_1, \ldots, !B_n, C_1, \ldots, C_m \longrightarrow B.$$

Given this style of sequent, it is natural to make a second modification to linear logic by introducing two kinds of implications: the linear implication, for which the right-introduction rule adds its assumption to the bounded part of a context, and the intuitionistic implication (written $\Rightarrow$), for which the right-introduction rule adds its assumption to the unbounded part of a context. Of course, the intended meaning of $B \Rightarrow C$ is $(!B) \multimap C$.

$$\frac{}{B \longrightarrow B}\ identity \qquad \frac{}{\Delta \longrightarrow \top}\ \top R$$

$$\frac{\Delta, B_i \longrightarrow C}{\Delta, B_1 \,\&\, B_2 \longrightarrow C}\ \&L\ (i = 1, 2) \qquad \frac{\Delta \longrightarrow B \qquad \Delta \longrightarrow C}{\Delta \longrightarrow B \,\&\, C}\ \&R$$

$$\frac{\Delta_1 \longrightarrow B \qquad \Delta_2, C \longrightarrow E}{\Delta_1, \Delta_2, B \multimap C \longrightarrow E}\ \multimap L \qquad \frac{\Delta, B \longrightarrow C}{\Delta \longrightarrow B \multimap C}\ \multimap R$$

$$\frac{\Delta, B_1, B_2 \longrightarrow C}{\Delta, B_1 \otimes B_2 \longrightarrow C}\ \otimes L \qquad \frac{\Delta_1 \longrightarrow B \qquad \Delta_2 \longrightarrow C}{\Delta_1, \Delta_2 \longrightarrow B \otimes C}\ \otimes R$$

$$\frac{\Delta \longrightarrow C}{\Delta, !\,B \longrightarrow C}\ !\,W \qquad \frac{\Delta, !\,B, !\,B \longrightarrow C}{\Delta, !\,B \longrightarrow C}\ !\,C \qquad \frac{\Delta, B \longrightarrow C}{\Delta, !\,B \longrightarrow C}\ !\,D \qquad \frac{!\,\Delta \longrightarrow B}{!\,\Delta \longrightarrow !\,B}\ !\,R$$

$$\frac{\Delta, B[t/x] \longrightarrow C}{\Delta, \forall x.B \longrightarrow C}\ \forall L \qquad \frac{\Delta \longrightarrow B[y/x]}{\Delta \longrightarrow \forall x.B}\ \forall R,$$

provided that y is not free in the lower sequent.

Figure 4: The proof system LL for a fragment of linear logic

$$\frac{\Delta \longrightarrow B \qquad \Delta', B \longrightarrow C}{\Delta, \Delta' \longrightarrow C}\ cut$$

Figure 5: A cut-rule for LL.

So far, only the logical connectives $\multimap$ and $\Rightarrow$ have been motivated. Consider a sequent in which the bounded formulas are atomic. If the only logical connectives are $\multimap$ and $\Rightarrow$ then every formula in the bounded part of the context must be used exactly once: that is, they must be accounted for in some *identity* inference rule by matching them with the same formula on the right of a sequent. Such rigid control of resources is limiting for most uses. For example, if a data base is held in the bounded part of a context, then querying the data base about an item makes that item unavailable elsewhere. Also, before a computation on the data base can be finished, it is necessary to "read" all items in this way. If, however, we add the connectives $\top$ and $\&$, we have the ability to erase parts of the bounded context (using $\top$) and to duplicate bounded contexts (using $\&$). Thus, non-destructively reading a value from a data base can be achieved by first making a copy of the data base from which we destructively read one item and delete the rest: the original data base is untouched. See Section 5.6 for an example of this kind of data base program specification.

Figure 6 presents a proof system $\mathcal{L}$ for the logic connectives $\top, \&, \multimap, \Rightarrow$, and $\forall$. We write $\Gamma; \Delta \vdash_{\mathcal{L}} B$ if the sequent $\Gamma; \Delta \longrightarrow B$ has a proof in $\mathcal{L}$. Notice that the bounded part of the left premise of the $\Rightarrow L$ inference rule is empty: this follows from the structure of an LL-proof with an application of $\multimap L$ to a formula of the form $!B \multimap C$. Notice as well that we assume without loss of generality that the *identity* inference of this system applies only where the right-hand side is an atomic formula. This technical restriction is used in the proof of Proposition 4 below.

Figure 7 presents the two cut rules for $\mathcal{L}$. Girard's proof of the cut-elimination theorem for linear logic (Girard, 1987a) can be adjusted to show that these two cut rules are admissible over $\vdash_{\mathcal{L}}$. A direct proof of cut elimination for this system can be found in the first author's dissertation, which also includes complete proofs of the other proof-theoretic propositions in this paper (Hodas,

$$\frac{}{\Gamma; A \longrightarrow A}\ identity \qquad \frac{\Gamma, B; \Delta, B \longrightarrow C}{\Gamma, B; \Delta \longrightarrow C}\ absorb \qquad \frac{}{\Gamma; \Delta \longrightarrow \top}\ \top R$$

$$\frac{\Gamma; \Delta, B_i \longrightarrow C}{\Gamma; \Delta, B_1 \mathbin{\&} B_2 \longrightarrow C}\ \& L \qquad \frac{\Gamma; \Delta \longrightarrow B \qquad \Gamma; \Delta \longrightarrow C}{\Gamma; \Delta \longrightarrow B \mathbin{\&} C}\ \& R$$

$$\frac{\Gamma; \Delta_1 \longrightarrow B \qquad \Gamma; \Delta_2, C \longrightarrow E}{\Gamma; \Delta_1, \Delta_2, B \multimap C \longrightarrow E}\ \multimap L \qquad \frac{\Gamma; \Delta, B \longrightarrow C}{\Gamma; \Delta \longrightarrow B \multimap C}\ \multimap R$$

$$\frac{\Gamma; \emptyset \longrightarrow B \qquad \Gamma; \Delta, C \longrightarrow E}{\Gamma; \Delta, B \Rightarrow C \longrightarrow E}\ \Rightarrow L \qquad \frac{\Gamma, B; \Delta \longrightarrow C}{\Gamma; \Delta \longrightarrow B \Rightarrow C}\ \Rightarrow R$$

$$\frac{\Gamma; \Delta, B[t/x] \longrightarrow C}{\Gamma; \Delta, \forall x.B \longrightarrow C}\ \forall L \qquad \frac{\Gamma; \Delta \longrightarrow B[y/x]}{\Gamma; \Delta \longrightarrow \forall x.B}\ \forall R$$

provided that y is not free in the lower sequent.

Figure 6: $\mathcal{L}$: A proof system for the connectives $\top$, $\&$, $\multimap$, $\Rightarrow$, and $\forall$. The formula A in the *identity* rule is restricted, for convenience only, to be atomic.

$$\frac{\Gamma'; \Delta_1 \longrightarrow B \qquad \Gamma; \Delta_2, B \longrightarrow C}{\Gamma'; \Delta_1, \Delta_2 \longrightarrow C}\ cut \qquad \frac{\Gamma'; \emptyset \longrightarrow B \qquad \Gamma, B; \Delta \longrightarrow C}{\Gamma'; \Delta \longrightarrow C}\ cut!$$

Figure 7: Two forms of the cut rule for $\mathcal{L}$. Both rules have the proviso that $\Gamma \subseteq \Gamma'$.

1993). This particular presentation of the cut-rules will be useful in Section 6 when we characterize their admissibility in Proposition 9.

Proposition 3 *Let B be a formula, Γ a set of formulas, and Δ a multiset of formulas, all over the logical constants $\top, \&, \multimap, \Rightarrow$, and $\forall$. Let $B^\diamond$ be the result of repeatedly replacing all occurrences of $C_1 \Rightarrow C_2$ in B with $(!\,C_1) \multimap C_2$. (Applying $^\diamond$ to a set or multiset of formulas results in the multiset of $^\diamond$ applied to each member.) Then $\Gamma; \Delta \vdash_{\mathcal{L}} B$ if and only if $!(\Gamma^\diamond), \Delta^\diamond \vdash_{LL} B^\diamond$.*

The proof in each direction can be shown by presenting a simple transformation between proofs in the two proof systems.

Proposition 4 *Let B be a formula, Γ a set formulas, and Δ a multiset of formulas all over the logical connectives $\top, \&, \multimap, \Rightarrow$, and $\forall$. The sequent $\Gamma; \Delta \longrightarrow B$ has a proof in $\mathcal{L}$ if and only if it has a uniform proof in $\mathcal{L}$.*

Proof. In the reverse direction, the proof is immediate, since a uniform $\mathcal{L}$-proof is certainly an $\mathcal{L}$-proof. In the forward direction we provide a non-deterministic algorithm that converts an arbitrary $\mathcal{L}$-proof to a uniform $\mathcal{L}$-proof of the same endsequent. We define a *non-uniform rule occurrence* as any occurrence of a left-rule in which the right-hand side of the conclusion is not an atomic formula. We also note that the *absorb* rule is considered a left-rule. The *rank* of a non-uniform rule occurrence is the height of the subproof of the right-hand premise if the rule occurrence is either $\multimap L$ or $\Rightarrow L$ or is the height of the subproof of the sole premise for any other left-hand rule occurrence. The algorithm is given as follows:

1. If the proof is uniform, terminate; otherwise, go to the next step.

2. Select a non-uniform rule occurrence with the property that the sub-proof(s) rooted at premise(s) of the rule are uniform. (That such a choice can be made is immediate given the assumption that *identity* inferences have only atomic right-hand sides.) Let C be the non-atomic right-hand side of the conclusion of this non-uniform rule occurrence.

3. One premise of the rule selected will be the conclusion of a right-rule that introduces the logical constant for C. There are only 25 such combinations of left-rules below right-introduction rules possible in an $\mathcal{L}$-proof. For all of these cases, it can be checked that the left-rule permutes up through the right rule. We will only illustrate one case, where an instance of $\multimap L$ occurs below an instance of $\& R$, as displayed below:

$$\dfrac{\begin{array}{c}\Xi_1\\ \Gamma;\Delta_1 \longrightarrow B_1\end{array} \quad \dfrac{\begin{array}{c}\Xi_2\\ \Gamma;\Delta_2,B_2 \longrightarrow C_1\end{array} \quad \begin{array}{c}\Xi_3\\ \Gamma;\Delta_2,B_2 \longrightarrow C_2\end{array}}{\Gamma;\Delta_2,B_2 \longrightarrow C_1 \,\&\, C_2}\,\& R}{\Gamma;\Delta_1,\Delta_2,B_1 \multimap B_2 \longrightarrow C_1 \,\&\, C_2}\,\multimap L$$

where we assume Ξ_1, Ξ_2, and Ξ_3 are uniform proofs of their respective endsequents. This proof structure is converted to one of the form:

$$\dfrac{\dfrac{\begin{array}{c}\Xi_1\\ \Gamma;\Delta_1 \longrightarrow B_1\end{array} \quad \begin{array}{c}\Xi_2\\ \Gamma;\Delta_2,B_2 \longrightarrow C_1\end{array}}{\Gamma;\Delta_1,\Delta_2,B_1 \multimap B_2 \longrightarrow C_1}\,\multimap L \quad \dfrac{\begin{array}{c}\Xi_1\\ \Gamma;\Delta_1 \longrightarrow B_1\end{array} \quad \begin{array}{c}\Xi_3\\ \Gamma;\Delta_2,B_2 \longrightarrow C_2\end{array}}{\Gamma;\Delta_1,\Delta_2,B_1 \multimap B_2 \longrightarrow C_2}\,\multimap L.}{\Gamma;\Delta_1,\Delta_2,B_1 \multimap B_2 \longrightarrow C_1 \,\&\, C_2}\,\& R$$

At this point, it is necessary to call this procedure recursively on the sub-proofs rooted at the premises of the new final rule, since new non-uniform rule occurrences may have been created immediately above. Fortunately, since such new occurrences of non-uniform rules will have strictly smaller ranks, this recursion will terminate. With the termination of the recursion(s), the number of non-uniform rule occurrences in the overall proof has been reduced by one.

4. Go to step 1.

Since each pass through the outer-loop of this algorithm reduces by one the number of non-uniform rule occurrences, the algorithm yields a uniform proof when it terminates. Thus, any sequent provable in $\mathcal{L}$ can be proved by a uniform $\mathcal{L}$-proof. ∎

Let $\mathcal{N}_1$ be the set of all first-order formulas over the logical connectives $\top, \&, \multimap, \Rightarrow$, and $\forall$. It follows immediately from Proposition 4 that the triple $\langle \mathcal{N}_1, \mathcal{N}_1, \vdash_{\mathcal{L}} \rangle$ is an abstract logic programming language. (Here, we assume that formulas in $\mathcal{N}_1$ can occur in both the bounded and unbounded parts of a sequent's left-hand side.)

As with system $\mathcal{I}'$ it is possible to restrict uniform proofs even further in the sense that the use of left-hand rules can be restricted to a form of backchaining. Consider the following definition. Let the syntactic variable B range over the logical formulas containing just the connectives $\top, \&, \multimap, \Rightarrow$, and $\forall$. Then $\|B\|$ is the smallest set of triples of the form $\langle \Gamma, \Delta, B' \rangle$ where Γ is a set of formulas and Δ is a multiset of formulas, such that

1. $\langle \emptyset, \emptyset, B \rangle \in \|B\|$,

2. if $\langle \Gamma, \Delta, B_1 \,\&\, B_2 \rangle \in \|B\|$ then both $\langle \Gamma, \Delta, B_1 \rangle \in \|B\|$ and $\langle \Gamma, \Delta, B_2 \rangle \in \|B\|$,

3. if $\langle \Gamma, \Delta, \forall x.B' \rangle \in \|B\|$ then for all closed terms t, $\langle \Gamma, \Delta, B'[t/x] \rangle \in \|B\|$,

$$\frac{\Gamma;\emptyset \longrightarrow B_1 \quad \ldots \quad \Gamma;\emptyset \longrightarrow B_n \quad \Gamma;\Delta_1 \longrightarrow C_1 \quad \ldots \quad \Gamma;\Delta_m \longrightarrow C_m}{\Gamma;\Delta_1,\ldots,\Delta_m, B \longrightarrow A}\ BC$$

provided $n, m \geq 0$, A is atomic, and $\langle\{B_1,\ldots,B_n\},\{C_1,\ldots,C_m\},A\rangle \in \|B\|$.

Figure 8: Backchaining for the proof system $\mathcal{L}$.

4. if $\langle\Gamma,\Delta,B_1 \Rightarrow B_2\rangle \in \|B\|$ then $\langle\Gamma \cup \{B_1\},\Delta,B_2\rangle \in \|B\|$, and
5. if $\langle\Gamma,\Delta,B_1 \multimap B_2\rangle \in \|B\|$ then $\langle\Gamma,\Delta \uplus \{B_1\},B_2\rangle \in \|B\|$. (Here, $\uplus$ denotes multiset union.)

Let $\mathcal{L}'$ be the proof system that results from replacing the *identity*, $\multimap L$, $\Rightarrow L$, $\& L$, and $\forall L$ rules in Figure 6 with the *backchaining* inference rule in Figure 8.

Proposition 5 *Let B be a formula, Γ a set of formulas, and Δ a multiset of formulas, all over the logical constants $\top, \&, \multimap, \Rightarrow$, and $\forall$. The sequent $\Gamma;\Delta \longrightarrow B$ has a proof in $\mathcal{L}$ if and only if it has a proof in $\mathcal{L}'$.*

Proof. In the reverse direction it is easy to show that each occurrence of a BC rule in $\mathcal{L}'$ can be converted to (possibly) several occurrences of the $\& L$, $\multimap L$, $\Rightarrow L$, $\forall L$, and *identity* rules in $\mathcal{L}$. The proof in the forward direction is more involved.

Let Ξ be an $\mathcal{L}$-proof of $\Gamma;\Delta \longrightarrow B$. Mark certain occurrences of formulas in the bounded part of some sequents in Ξ as follows. The unique formula in the bounded part of the conclusion of every *identity* rule is marked. By referring to Figure 6 we then mark additional formula occurrences using induction on the structure of proofs as follows:

- If the B_i formula occurrence in the $\& L$ rule is marked, then mark the occurrence of $B_1 \& B_2$ in its conclusion.
- If the $B[t/x]$ formula occurrence in the $\forall L$ rule is marked, then mark the occurrence of $\forall x.B$ in its conclusion.
- If the C formula occurrence in the right-hand premise of the $\multimap L$ rule is marked, then mark the occurrence of $B \multimap C$ in its conclusion
- If the C formula occurrence in the right-hand premise of the $\Rightarrow L$ rule is marked, then mark the occurrence of $B \Rightarrow C$ in its conclusion.

As in (Miller, 1989b), an occurrence of a left-introduction rule is called *simple* if the occurrence of the formula containing the logical connective introduced is marked. A uniform proof in which all occurrences of left-introduction rules are simple is called a *simple proof.*

Now observe two facts about simple proofs. First, if Ξ is simple, then Ξ can be transformed directly into an $\mathcal{L}'$-proof: simply collapse all chains of left-introduction rules (following the marking process) into one BC inference rule. Second, by permuting inference rules, any uniform $\mathcal{L}$-proof can be transformed into a simple proof. The proof of this is similar to the proof of Proposition 4. Find a non-simple occurrence of a left-introduction rule for which the subproofs of its premise(s) are simple proofs. One of the premises of this non-simple occurrence of a left-introduction rule must also be a left-introduction rule. Permute these two left-introduction rules. Consider, for example, the following case where these two left-introduction rules are $\multimap L$.

$$\dfrac{\begin{array}{c}\Xi_1\\ \Gamma;\Delta_1 \longrightarrow B_1\end{array} \quad \dfrac{\begin{array}{c}\Xi_2\\ \Gamma;\Delta_2,B_2 \longrightarrow C_1\end{array} \quad \begin{array}{c}\Xi_3\\ \Gamma;\Delta_3,C_2 \longrightarrow E\end{array}}{\Gamma;\Delta_2,\Delta_3,B_2,C_1 \multimap C_2 \longrightarrow E}\multimap L}{\Gamma;\Delta_1,\Delta_2,\Delta_3,B_1 \multimap B_2, C_1 \multimap C_2 \longrightarrow E}\multimap L$$

Here we assume Ξ_1, Ξ_2, and Ξ_3 are simple proofs. This proof structure is then converted to the following proof by permuting these two inference rule occurrences.

$$\dfrac{\dfrac{\begin{array}{c}\Xi_1\\ \Gamma;\Delta_1 \longrightarrow B_1\end{array} \quad \begin{array}{c}\Xi_2\\ \Gamma;\Delta_2,B_2 \longrightarrow C_1\end{array}}{\Gamma;\Delta_1,\Delta_2,B_1 \multimap B_2 \longrightarrow C_1}\multimap L \quad \begin{array}{c}\Xi_3\\ \Gamma;\Delta_3,C_2 \longrightarrow E\end{array}}{\Gamma;\Delta_1,\Delta_2,\Delta_3,B_1 \multimap B_2, C_1 \multimap C_2 \longrightarrow E}\multimap L$$

It may be necessary to continue permuting inference rules in this fashion since, in this case, the subproof of the sequent $\Gamma;\Delta_1,\Delta_2,B_1 \multimap B_2 \longrightarrow C_1$ may not be simple. The result of continuing this process is then a simple proof of the sequent $\Gamma;\Delta_1,\Delta_2,\Delta_3,B_1 \multimap B_2, C_1 \multimap C_2 \longrightarrow E$. In this way, all non-simple occurrences of left-introduction rules can be eliminated, giving rise to a simple proof, which can, as noted, be converted to an $\mathcal{L}'$ proof. ∎

Note that, unlike the system $\mathcal{I}'$, proofs in $\mathcal{L}'$ are not necessarily uniform due to the presence of the *absorb* rule, which may act on sequents with non-atomic right-hand sides. Nevertheless, it is easy to see that all uses of this rule can be pushed up a proof tree so that they occur only immediately below instances of *BC*. Such $\mathcal{L}'$ proofs are then uniform.

As was noticed in Section 2, since we are only interested in cut-free proofs, it is possible to permit different sets of formulas to occur on the left and right of the sequent arrow. As in Section 2, there are at least two ways to do this. We can expand the logic by allowing some occurrences of additional logical constants, or we can use higher-order quantification with respect to the given logic to "define" the additional constants.

Using the first approach, consider the following definition of two classes of formulas over the logical constants $\top, \mathbf{1}, \&, \otimes, \oplus, \multimap, \Rightarrow, !, \forall$, and $\exists$.

$$R := \top \mid A \mid R_1 \,\&\, R_2 \mid G \multimap R \mid G \Rightarrow R \mid \forall x.R$$
$$G := \top \mid A \mid G_1 \,\&\, G_2 \mid R \multimap G \mid R \Rightarrow G \mid \forall x.G \mid G_1 \oplus G_2 \mid \mathbf{1} \mid G_1 \otimes G_2 \mid \,!G \mid \exists x.G$$

Here, R-formulas, called *resource formulas*, can appear in either part of the proof context (on the left of a sequent) while G-formulas, called *goal formulas*, can appear on the right of sequents. Given this extension, it is necessary to add to the proof system $\mathcal{L}'$ right-introduction rules for $\mathbf{1}, \oplus, \otimes, !$ and $\exists$. Let $\mathcal{L}''$ be the proof system that results from adding the right-introduction rules in Figure 9 to $\mathcal{L}'$. (Notice that since $\mathbf{1}$ is logically equivalent to $!\top$, it could be dropped from this definition.) In this same setting, it is also possible to use a more restrictive definition for resource formulas (R-formulas):

$$R := \top \mid A \mid R_1 \,\&\, R_2 \mid G \multimap A \mid G \Rightarrow A \mid \forall x.R. \qquad (*)$$

Although such a simplification does not change the expressiveness of the logic much, it makes the presentation of an interpreter for the logic simpler, as will be seen in Section 7.

The second approach does not extend the logic by adding these logical constants directly but instead axiomatizes their right-introduction rules using higher-order quantification. The following clauses are appropriate definitions for these constants:

$$\frac{}{\Gamma;\emptyset \longrightarrow \mathbf{1}}\ \mathbf{1}R \qquad \frac{\Gamma;\emptyset \longrightarrow B}{\Gamma;\emptyset \longrightarrow\ !B}\ !R \qquad \frac{\Gamma;\Delta \longrightarrow B_i}{\Gamma;\Delta \longrightarrow B_1 \oplus B_2}\ \oplus R\ (i=1,2)$$

$$\frac{\Gamma;\Delta \longrightarrow B[x/t]}{\Gamma;\Delta \longrightarrow \exists x.B}\ \exists R \qquad \frac{\Gamma;\Delta_1 \longrightarrow B_1 \quad \Gamma;\Delta_2 \longrightarrow B_2}{\Gamma;\Delta_1,\Delta_2 \longrightarrow B_1 \otimes B_2}\ \otimes R$$

Figure 9: Additional rules for positive occurrences of $\mathbf{1}, \otimes, \oplus, !$, and $\exists$.

$$\begin{gathered}
\forall P \forall Q[P \multimap (P \oplus Q)] \\
\forall P \forall Q[Q \multimap (P \oplus Q)] \\
\forall B \forall T[(B\ T) \multimap (\exists B)] \\
\top \Rightarrow \mathbf{1} \\
\forall P \forall Q[P \multimap Q \multimap (P \otimes Q)] \\
\forall P[P \Rightarrow\ !P]
\end{gathered}$$

If we assume that there are no negative occurrences of any of these constants within a proof (except in these defining formulas) then this amounts to the same restriction as in the first approach.

4 An embedding of hereditary Harrop formulas

Girard has presented a mapping of intuitionistic logic into linear logic that preserves not only provability but also proofs (Girard, 1987a). On the fragment of intuitionistic logic containing *true*, $\wedge$, $\supset$, and $\forall$, the translation is given by:

$$\begin{gathered}
(A)^0 = A, \text{ where } A \text{ is atomic}, \\
(true)^0 = \mathbf{1}, \\
(B_1 \wedge B_2)^0 = (B_1)^0 \,\&\, (B_2)^0, \\
(B_1 \supset B_2)^0 = !(B_1)^0 \multimap (B_2)^0, \\
(\forall x.B)^0 = \forall x.(B)^0.
\end{gathered}$$

However, if we are willing to focus attention on only cut-free proofs in $\mathcal{I}'$ and $\mathcal{L}''$, it is possible to define a "tighter" translation. Consider the following two translation functions.

$$\begin{gathered}
(A)^+ = (A)^- = A, \text{ where } A \text{ is atomic} \\
(true)^+ = \mathbf{1} \\
(true)^- = \top \\
(B_1 \wedge B_2)^+ = (B_1)^+ \otimes (B_2)^+ \\
(B_1 \wedge B_2)^- = (B_1)^- \,\&\, (B_2)^- \\
(B_1 \supset B_2)^+ = (B_1)^- \Rightarrow (B_2)^+ \\
(B_1 \supset B_2)^- = (B_1)^+ \multimap (B_2)^- \\
(\forall x.B)^+ = \forall x.(B)^+ \\
(\forall x.B)^- = \forall x.(B)^-
\end{gathered}$$

If we allow positive occurrences of $\vee$ and $\exists$ within cut-free proofs, as in proofs involving the hereditary Harrop formulas, we would also need the following two clauses.

$$\begin{gathered}
(B_1 \vee B_2)^+ = (B_1)^+ \oplus (B_2)^+ \\
(\exists x.B)^+ = \exists x.(B)^+
\end{gathered}$$

Proposition 6 *Let B be a formula and Γ a set of formulas, all over the logical constants* $true, \wedge, \supset$, *and* $\forall$. *Define* $\Gamma^- = \{C^- \mid C \in \Gamma\}$. *Then,* $\Gamma \vdash_{\mathcal{I}} B$ *if and only if the sequent* $\Gamma^-; \emptyset \longrightarrow B^+$ *has a cut-free proof in* $\mathcal{L}''$.

Proof. First observe that if B is a formula over the logical constants $true, \wedge, \supset$, and $\forall$ then $\langle \Delta, A\rangle \in |B|$ if and only if $\langle \emptyset, \Delta^+, A\rangle \in \|B^-\|$. Let Ξ be an $\mathcal{I}'$-proof of $\Gamma \longrightarrow B$. This proof can be converted to a proof Ξ^o of $\Gamma^-; \emptyset \longrightarrow B^+$ by converting the inference rules $trueR$, $\wedge R$, $\supset R$, and $\forall R$ to the $\mathcal{L}''$ inference rules $\mathbf{1}R$, $\otimes R$, $\Rightarrow R$, and $\forall R$. Furthermore, instances of the BC rule of $\mathcal{I}'$ need to be converted to BC paired with *absorb* in $\mathcal{L}''$. For the converse, let Ξ^o be an $\mathcal{L}''$-proof of $\Gamma^-; \emptyset \longrightarrow B^+$. As was mentioned in the last section, we can assume that the only occurrences of the *absorb* rule are such that their premise is the conclusion of an instance of the BC rule. Such a proof can be converted to a proof in $\mathcal{L}''$ by reversing the conversion mentioned for the first case. (Girard has pointed out to us that this proposition should be provable directly within his LU proof system (Girard, 1991).) ∎

A consequence of this proposition is that $\mathcal{I}'$-proofs involving Horn clauses or hereditary Harrop formulas are essentially the same as the $\mathcal{L}''$-proofs of their translations. This suggests how to design the concrete syntax of a linear logic programming language so that the interpretation of Prolog and λProlog programs remains unchanged when embedded into this new setting. For example, the Prolog syntax

$$A_0 \;:-\; A_1, \ldots, A_n$$

is traditionally intended to denote (the universal closure of) the formula

$$(A_1 \wedge \ldots \wedge A_n) \supset A_0.$$

Given the negative translation above, such a Horn clause would then be translated to the linear logic formula

$$(A_1 \otimes \ldots \otimes A_n) \multimap A_0.$$

Thus, the comma in Prolog denotes $\otimes$ and $:-$ denotes the converse of $\multimap$.

For another example, the natural deduction rule for implication-introduction, often expressed using the diagram

$$\frac{\begin{array}{c}(A)\\ \vdots \\ B\end{array}}{A \supset B,}$$

can be written as the following first-order formula for axiomatizing a truth predicate (see Felty, 1993; Felty and Miller, 1988; Paulson, 1990):

$$\forall A \forall B((true(A) \supset true(B)) \supset true(A\ imp\ B)),$$

where the domain of quantification is over propositional formulas of the object-language and *imp* is the object-level implication. This formula is written in λProlog using the syntax

```
true (A imp B)  :-  true A => true B.
```

Given the above proposition, this formula can be translated to the formula

$$\forall A \forall B((true\ A \Rightarrow true\ B) \multimap true\ (A\ imp\ B)),$$

which means that the λProlog symbol `=>` should denote $\Rightarrow$. Thus, in the implication introduction rule displayed above, the meta-level implication represented as three vertical dots can be interpreted as an intuitionistic implication while the meta-level implication represented as the horizontal bar can be interpreted as a linear implication.

Notice that when building $\mathcal{L}''$-proofs of sequents translated from intuitionistic logic, the bounded part of sequents is non-empty only at the point that the backchaining rule is applied. If we relate this observation to the construction of normal λ-terms, where backchaining corresponds to the application of a typed constant or variable, we draw the (obvious) conclusion that every normal λ-term has exactly one head symbol.

In order to present several examples in the next section, we shall make use of the following syntactic conventions for specifying resource formulas and goal formulas. These conventions are motivated by the last proposition so that the syntax of Prolog and λProlog embed naturally into the extended language. The symbols `,` (comma), `true`, `=>`, and `:-` of Prolog and λProlog will be used here to represent $\otimes$, $\mathbf{1}$, $\Rightarrow$, and the converse of $\multimap$, respectively. In addition, we allow formulas to have occurrences of `&`, `bang`, `erase`, `-o`, and `<=`, which denote, respectively, $\&$, $!$, $\top$, $\multimap$, and the converse of $\Rightarrow$. We shall also adopt the standard convention that a token with an initial upper case letter that is not explicitly quantified in a formula is intended to be universally quantified (respectively, existentially quantified) around a resource formula (goal formula) with outermost scope. Finally, the clauses of a program are assumed to reside in the unbounded portion of an initial proof context.

5 Some example programs

5.1 Context management in theorem provers

Intuitionistic logic is a useful meta-logic for the specification of provability in various object-logics. For example, consider axiomatizing provability in propositional, intuitionistic logic over the logical symbols `imp`, `and`, `or`, and `false` (denoting object-level implication, conjunction, disjunction, and absurdity). A reasonable specification of the natural deduction inference rule for implication introduction is:

```
pv (A imp B) :- hyp A => pv B.
```

where `pv` and `hyp` are meta-level predicates denoting provability and hypothesis. (This specification of implication introduction is similar to that given in the preceding section.) Operationally, this formula states that one way to prove `A imp B` is to add the object-level hypothesis `A` to the context and attempt a proof of `B`. In the same setting, conjunction elimination can be expressed by the formula

```
pv G :- hyp (A and B), (hyp A => hyp B => pv G).
```

This formula states that in order to prove some object-level formula `G`, first check to see if there is a conjunctive hypothesis, say `(A and B)`, in the context and, if so, attempt a proof of `G` from the context extended with the two hypotheses `A` and `B`. Other introduction and elimination rules can be specified similarly. Finally, the formula

```
pv G :- hyp G.
```

is needed to actually complete a proof. With the complete specification, it is easy to prove that there is a proof of `(pv G)` from the assumptions `(hyp H1)`, ..., `(hyp Hi)` in the meta-logic if and only if there is a proof of `G` from the assumptions `H1`, ..., `Hi` in the object-logic.

```
pv (A and B) :- pv A & pv B.
pv (A imp B) :- hyp A -o pv B.
pv (A or  B) :- pv A.
pv (A or  B) :- pv B.
pv G :- hyp (A and B), (hyp A -o hyp B -o pv G).
pv G :- hyp (A or  B), ((hyp A -o pv G) & (hyp B -o pv G)).
pv G :- hyp (C imp B), ((hyp (C imp B) -o pv C) &
                        (hyp B -o pv G)).
pv G :- hyp false, erase.
pv G :- hyp G, erase.
```

Figure 10: A specification of an intuitionistic propositional object-logic

Unfortunately, an intuitionistic meta-logic does not permit the natural specification of provability in logics that have restricted contraction rules — such as linear logic itself — because hypotheses are maintained in intuitionistic logic contexts and hence can be used zero or more times. Even in describing provability for propositional intuitionistic logic there are some drawbacks. For instance, it is not possible to logically express the fact that a conjunctive or disjunctive formula in the proof context needs to be eliminated at most once. So, for example, in the specification of conjunction elimination, once the context is augmented with the two conjuncts, the conjunction itself is no longer needed in the context.

If, however, we replace the intuitionistic meta-logic with our refinement based on linear logic, these observations about use and re-use in intuitionistic logic can be specified elegantly, as is done in Figure 10. In that specification, a hypothesis is both "read from" and "written into" a context during the elimination of implications. All other elimination rules simply "read from" the context; they do not "write back." The formulas represented by the last two clauses in Figure 10 use a $\otimes$ with $\top$: this allows for all unused hypotheses to be erased, since the object logic has no restrictions on weakening.

It should be noted that this specification cannot be used effectively with a depth-first interpreter because if the implication left rule can be used once, it can be used any number of times, thereby causing the interpreter to loop. Fortunately, improvements in the implication left-introduction rule are known. For example, the proof system given by Dyckhoff in (Dyckhoff, 1992) can be expressed directly in this setting by replacing the one formula specifying implication elimination in Figure 10 with the five clauses for implication elimination and the (partial) axiomatization of object-level atomic formulas in Figure 11. Executing this linear logic program in a depth-first interpreter yields a decision procedure for propositional intuitionistic logic.

5.2 Toggling a switch

If we assume that the state of a switch is stored in the bounded part of the proof context using one of the atomic formulas `on` or `off`, then the following two clauses specify a higher-order predicate `toggle` that is provable of any formula `G` in a given context if `G` is provable when the switch is set to the opposite setting.

```
toggle G   :-  on,  (off -o G).
toggle G   :-  off, (on  -o G).
```

While this example involves a quantification over propositions (the variable `G`), and as such is not strictly a first-order specification, the intended meaning of the specification should be clear.

```
pv G :- hyp ((C imp D) imp B),
          ((hyp (D imp B) -o pv (C imp D)) & (hyp B -o pv G)).
pv G :- hyp ((C and D) imp B), (hyp (C imp (D imp B)) -o pv G).
pv G :- hyp ((C or  D) imp B), (hyp (C imp B) -o hyp (D imp B) -o pv G).
pv G :- hyp (false imp B), pv G.
pv G :- hyp (A imp B), isatom A, hyp A, (hyp B -o hyp A -o pv G).

isatom p.
isatom q.
isatom r.
```

Figure 11: A contraction-free formulation of $\supset L$

$$\dfrac{\dfrac{}{\Gamma; \mathtt{off} \longrightarrow \mathtt{off}} \qquad \dfrac{\begin{array}{c}\vdots \\ \Gamma; \Delta, \mathtt{on} \longrightarrow \mathtt{G}\end{array}}{\Gamma; \Delta \longrightarrow \mathtt{on} \multimap \mathtt{G}}}{\dfrac{\Gamma; \Delta, \mathtt{off} \longrightarrow \mathtt{off} \otimes (\mathtt{on} \multimap \mathtt{G})}{\Gamma; \Delta, \mathtt{off} \longrightarrow \mathtt{toggle}\ G}}$$

Figure 12: Proof search for toggling a switch

Figure 12 (in which the set Γ is assumed to contain the above two clauses for `toggle`) shows how a bottom-up search using these clauses might progress.

This example illustrates an approach which has previously been used by the authors to provide a notion of object state in object-oriented logic programming (Hodas and Miller, 1990). The linear refinement of hereditary Harrop formulas provides a straightforward declarative treatment of state update in that setting.

5.3 Permuting a list

Since the bounded part of contexts in $\mathcal{L}$-proofs are intended to be multisets and not lists, it is a simple matter to permute a list by first loading the list's members into the bounded part of a context and then unloading them. The latter operation is nondeterministic and can succeed once for each permutation of the loaded list. Consider the following simple program:

```
load nil K      :- unload K.
load (X::L) K   :- (item X -o load L K).
unload nil.
unload (X::L)   :- item X, unload L.
```

Here, `nil` denotes the empty list and `::` the list constructor. The meaning of `load` and `unload` is dependent on the contents of the bounded part of the context, so the correctness of these clauses must be stated relative to a context. Let Γ be a set of formulas containing the four formulas displayed above and any other formulas that do not contain either `item`, `load`, or `unload` as their head symbol. (The *head symbol* of a formula of the form A or $G \multimap A$ is the predicate symbol that is the head of the atom A.) Let Δ be the multiset containing exactly the atomic formulas

$$\mathtt{item}\ a_1,\ \ldots,\ \mathtt{item}\ a_n.$$

We shall say that such a context *encodes* the multiset $\{a_1, \ldots, a_n\}$. It is now an easy matter to prove the following two assertions about `load` and `unload`:

- The goal `(unload K)` is provable from $\Gamma; \Delta$ if and only if `K` is a list containing the same elements with the same multiplicity as the multiset encoded in Δ.
- The goal `(load L K)` is provable from $\Gamma; \Delta$ if and only if `K` is a list containing the same elements with the same multiplicity as in the list `L` together with the multiset encoded in the context Δ.

In order for `load` and `unload` to correctly permute the elements of a list, we must guarantee two things about the context: first, the predicates `item`, `load`, and `unload` cannot be used as head symbols in any part of the context except as specified above and, second, the bounded part of a context must be empty at the start of the computation of a permutation. It is possible to handle the first condition by making use of appropriate universal quantifiers over the predicate names `item`, `load`, and `unload`. Such an approach to the lexical scoping of names has been addressed in depth in previous papers (Miller, 1989a, 1990) and will not be taken up here. The second condition — that the unbounded part of a context is empty — can be managed by making use of the modal nature of !, which we now discuss in more detail.

5.4 The modality of !

One extension to logic programming languages that has been studied for several years is the *demo*-predicate (Bowen and Kowalski, 1982). The intended meaning of attempting a query of the form $demo(R, G)$ in context Γ is simply attempting the query G in the context containing only R; that is, the main context is forgotten during the scope of the *demo*-predicate. The use of a !'ed goal has a related meaning.

Consider proving the sequent $\Gamma; \Delta \longrightarrow\ !G_1 \otimes G_2$, where Γ and Δ are composed of resource formulas and G_1 and G_2 are goal formulas. Given the completeness of uniform proofs for the system $\mathcal{L}''$, this is provable if and only if the two sequents $\Gamma; \emptyset \longrightarrow G_1$ and $\Gamma; \Delta \longrightarrow G_2$ are provable. In other words, the use of the "of-course" operator forces G_1 to be proved with an empty bounded context. Thus, with respect to bounded resources, the goal $!(R \multimap G)$ behaves similarly to $demo(R, G)$. In a sense, since bounded resources can come and go within contexts during a computation, they can be viewed as "contingent" resources, whereas unbounded resources are "necessary". The of-course operator attached to a goal ensures that the provability of the goal depends only on the necessary and not the contingent resources of the context.

It is now clear how to define the permutation of two lists given the example program above: add either the formula

```
perm L K  :-  bang(load L K).
```

or, equivalently, the formula

```
perm L K  <=  load L K.
```

to those defining `load` and `unload`. Thus attempting to prove `(perm L K)` will result in an attempt to prove `(load L K)` with an empty bounded context. From the description of `load` above, `L` and `K` must be permutations of each other.

5.5 Multiset rewriting

The ideas presented in the permutation example can easily be expanded upon to show how the bounded part of a context can be employed to do multiset rewriting. Let H be the *multiset rewriting system* $\{\langle L_i, R_i\rangle \mid i \in I\}$ where for each $i \in I$ (a finite index set), L_i and R_i are finite multisets. Define the relation $M \leadsto_H N$ on finite multisets to hold if there is some $i \in I$ and some multiset C such that M is $C \uplus L_i$ and N is $C \uplus R_i$. Let $\leadsto_H^*$ be the reflexive and transitive closure of $\leadsto_H$.

Given a rewriting system H, we wish to specify a binary predicate `rewrite` such that `(rewrite L K)` is provable if and only if the multisets encoded by `L` and `K` stand in the $\leadsto_H^*$ relation. Let Γ_0 be the following set of formulas (these are independent of H):

```
rewrite L K <= load L K.

load (X::L) K :- (item X -o load L K).
load nil    K :- rew K

rew K :- unload K.

unload (X::L) :- item X, unload L.
unload nil.
```

Taken alone, these clauses give a slightly different version of the `permute` program of the last example. The only addition is the binary predicate `rew`, which will be used as a socket into which we can plug a particular rewrite system.

In order to encode a rewrite system H, each rewrite rule in H is given by a formula specifying an additional clause for the `rew` predicate as follows: If H contains the pair $\langle\{a_1, \ldots, a_n\}, \{b_1, \ldots, b_m\}\rangle$ then this pair is encoded as the clause:

`rew K :- item` a_1`, ..., item` a_n`, (item` b_1 `-o (... -o (item` b_m `-o rew K)...)).`

If either n or m is zero, the appropriate portion of the formula is deleted. Operationally, this clause reads the a_i's out of the bounded context, loads the b_i's, and then attempts another rewrite. Let Γ_H be the set resulting from encoding each pair in H. As an example, if H is the set of pairs $\{\langle\{a, b\}, \{b, c\}\rangle, \langle\{a, a\}, \{a\}\rangle\}$ then Γ_H is the set of clauses:

```
rew K  :-   item a, item b, (item b -o (item c -o rew K)).
rew K  :-   item a, item a,            (item a -o rew K).
```

The following claim is easy to prove about this specification: if M and N are multisets represented as the lists `L` and `K`, respectively, then $M \leadsto_H^* N$ if and only if the goal `(rewrite L K)` is provable from the context $\Gamma_0, \Gamma_H; \emptyset$.

One drawback of this example is that `rewrite` is a predicate on lists, though its arguments are intended to represent multi-sets, and are operated on as such. Therefore, for each M, N pair this program generates a factor of at least $n!$ more proofs than the corresponding rewriting proofs, where n is the cardinality of the multiset N. This redundancy could be addressed either by implementing a data type for multi-sets or, perhaps, by investigating a non-commutative variant of linear logic.

5.6 A data base query language

A program that implements a simple data base query language is displayed in Figure 13. To make this example interesting, we have augmented the language with the `read`, `write`, and `nl` (new line)

```
db :- write "Command: ", read Command, do Command.
db :- write "Try again.", nl, db.

do (enter Entry)   :- entry Entry -o db.
do (commit Entry)  :- entry Entry => db.
do (retract Entry) :- entry Entry, db.
do (upd Old New)   :- entry Old, (entry New -o db).
do (check Q)       :- (entry Q, erase, write Q, write " is an entry.", nl) &
                      db.
do (necessary Q)   :- (bang (entry Q), erase, write Q,
                       write " is a necessary entry", nl) & db.
do quit            :- erase.
```

Figure 13: A simple data base query program

input/output commands. We shall also assume that the sub-goals of a conjunction are attempted in a left-to-right order. The data base is stored in both the bounded and unbounded parts of the context using the unary predicate `entry`. Entries in the bounded part can be retracted or updated; entries in the unbounded context are permanent and cannot be updated or retracted. A session using this program might proceed as follows:

```
Command: enter (enroll jane cs1).
Command: check (enroll jane X).
(enroll jane cs1) is an entry.
Command: upd (enroll jane cs1) (enroll jane cs2).
Command: check (enroll jane X).
(enroll jane cs2) is an entry.
Command: commit (student jane).
Command: enter (student bob).
Command: necessary(student X).
(student jane) is a necessary entry
Command: retract (student jane).
Command: necessary (student X).
(student jane) is a necessary entry
Command: necessary (student bob).
Try again.
Command: quit.
```

This example shows some limitations of linear contexts in this data base setting. For example, it does not seem possible to query a context to find out if an entry is contingent and not necessary (although accommodating negation-as-failure would make this possible). Thus it is not possible (with just this logic) to check if a command is attempting to retract a necessary (committed) entry: as seen in the sample session, such an operation is accepted but ineffective.

5.7 A gap-threading parser for English relative clauses

Our final example is a simple natural language parser which demonstrates how linear logic can be used to implement a technique known as *gap threading* (Pereira and Shieber, 1987). Intuitionistic contexts have been proposed as a means of managing the introduction and scoping of gaps

```
sent P1 P2          :- bang (np P1 P0), vp P0 P2.
vp P1 P2            :- tv P1 P0, np P0 P2.
vp P1 P2            :- stv P1 P0, sbar P0 P2.
sbar (that::P1) P2 :- sent P1 P2.
np P1 P2            :- pn P1 P2.
rel (whom::X) Y     :- all z\(np z z) -o sent X Y.
pn  (mary::L)      L.
pn  (bob::L)       L.
pn  (ann::L)       L.
tv  (loves::L)     L.
tv  (married::L)  L.
stv (believes::L) L.
```

Figure 14: A simple parser for gaps in English

(Pareschi, 1989; Pareschi and Miller, 1990). This approach, although modeling various aspects of gap-threading correctly, is unsatisfactory for at least two reasons. First, the restriction that a gap, once introduced, must be used is not easy to enforce using an intuitionistic context. Therefore, the phrase "whom Bob married Ann" would parse incorrectly as a valid relative clause. Second, various restrictions on where gaps may occur are not explained using intuitionistic contexts. For example, gaps introduced by "whom" can occur in object but not nominal positions. Thus the phrase "whom Ann believes that Bob married" is correct (the gap is the object of "married") while "whom Ann believes that married Bob" is incorrect (the gap is the subject of "married"). The "modal" distinction between these two kinds of noun phrases is not addressed naturally using intuitionistic logic.

The small logic program in Figure 14 is a simple parser based on *definite clause grammars (DCG)* (Pereira and Warren, 1980) extended with some uses of linear logic. Each category of the grammar, such as `sent` for sentence, `vp` for verb phrase, `sbar` for complement clauses, etc., is given two arguments, denoting a difference list of words. The rule for relative clauses (`rel`) introduces a gap by loading the formula `all z\(np z z)` (which denotes the logical expression $\forall z.(\mathtt{np}\ z\ z)$) into the bounded context. This formula represents a contingent resource: a noun phrase of zero length. Because of the restrictions placed on the management of the bounded context, the requirement that introduced gaps be consumed is handled by the logic.

The ! operator provides a simple mechanism for declaring that gaps cannot be consumed during the parsing of certain occurrences of parts of speech: for example, the subject noun phrase in the formula specifying `sent` is protected by a !. This blocks any contingent resources from being used in its proof. Thus, an introduced gap can be used to prove the noun phrase mentioned in the `vp` clause but not the one in the `sent` clause. Therefore, the two goals

```
rel (whom::ann::believes::that::bob::married::nil) nil.
rel (whom::bob::married::nil) nil.
```

are provable but the two goals

```
rel (whom::ann::believes::that::married::bob::nil) nil.
rel (whom::bob::married::ann::nil) nil.
```

are not. As this parser rules out subject extraction, sentences that require such extractions must be handled with additional specialized grammar rules. Several similar types of "island constraints" occur in natural language parsing problems (Pereira and Shieber, 1987). The use of !'ed formulas may aid in handling these constraints as well.

The duplication of gaps across conjunctions in such phrases as "the doctor whom Bob married and Jane knew" can be explained well using a & to copy gaps. To this end, the following formula can be add to the grammar above to handle the conjunction of sentences:

```
sent P1 P2 :- sent P1 (and::P3) & sent P3 P2.
```

The first author has extended this application of linear logic to natural language processing (Hodas, 1992). His dissertation also includes a set of extended examples along the lines of those given here (Hodas, 1993).

6 A model theoretic semantics

Besides the fact that logic programming languages can be characterized by their use of goal-directed search, they also generally share the characteristic that given a program, there exists a single model such that validity in that model is equivalent to provability from the program. Thus when designing a program (a proof context), a programmer can think of the meaning of a query operationally, as the search for uniform, cut-free proofs of the query in that context, or declaratively, as truth in the particular *canonical* model that the program represents.

The canonical model for first-order Horn clauses is well known (Apt and van Emden, 1982). A Kripke-like model construction was proposed in (Miller, 1989b) as the canonical model for a subset of first-order hereditary Harrop formulas, but unfortunately the construction given there was not shown to be an actual model in the sense of possible-world semantics. A canonical model was given in (Miller, 1992) for the logic programming language described in Section 2. We shall use the approach given in (Miller, 1992) to develop a canonical model for the logic programming language based on $\top$, $\&$, $\multimap$, and $\Rightarrow$. We shall only consider the propositional case here since most of the aspects of the model are illustrated in just that simple setting. The analysis of the quantificational case (including higher-type quantification) given in (Miller, 1992) applies equally well here.

Let $\langle R, +, 0\rangle$ be a commutative monoid and let $\langle \mathcal{W}, \leq\rangle$ be a partially ordered set. We shall call R the *monoid of bounded resources* and $\mathcal{W}$ the *set of possible worlds*. A *(propositional) Kripke interpretation* is an order preserving mapping from $\langle \mathcal{W}, \leq\rangle$ to the powerset of the set of atomic formulas (here, propositional letters). A *resource indexed model* $\mathcal{M}$ is an R-indexed set of Kripke interpretations, $\{K_r \mid r \in R\}$. Thus the set $K_r(w)$ is intended to denote the set of propositional formulas that are true at world $w \in \mathcal{W}$ in interpretation K_r.

Satisfaction in a structure $\mathcal{M} = \{K_r \mid r \in R\}$ is defined by the following induction on the structure of propositional formulas.

- $K_r, w \models \top$.
- $K_r, w \models A$ if A is atomic and $A \in K_r(w)$.
- $K_r, w \models B_1 \,\&\, B_2$ if $K_r, w \models B_1$ and $K_r, w \models B_2$.
- $K_r, w \models B_1 \multimap B_2$ if $\forall r' \in R, \forall w' \in \mathcal{W}$ if $w \leq w'$ and $K_{r'}, w' \models B_1$ then $K_{r+r'}, w' \models B_2$.
- $K_r, w \models B_1 \Rightarrow B_2$ if $\forall w' \in \mathcal{W}$ if $w \leq w'$ and $K_0, w' \models B_1$ then $K_r, w' \models B_2$.

Finally, we write $\mathcal{M} \models B$ if $\forall w \in \mathcal{W}$, $K_0, w \models B$. In a sense, the Kripke interpretation K_0 models truth in the usual intuitionistic sense while K_r models truth that has been moved out-of-phase (borrowing an image from Girard (Girard, 1987a)) by the presence of the resource $r \in R$. The following lemma is proved by a simple induction on the structure of propositional formulas.

Lemma 7 *Let $r \in R$ and $w, w' \in \mathcal{W}$. If $w \leq w'$ and $K_r, w \models B$ then $K_r, w' \models B$.*

To aid in describing the satisfaction of sequents, we add the following rules for determining the satisfaction of $\mathbf{1}$ and $\otimes$. If we were investigating models for a logic that allowed unrestricted occurrences of $\mathbf{1}$ and $\otimes$, these satisfaction rules would need to be replaced by stronger rules.

- $K_0, w \models \mathbf{1}$.
- $K_r, w \models B_1 \otimes B_2$ if there are $r_1, r_2 \in R$ such that $r = r_1 + r_2$ and $K_{r_1}, w \models B_1$ and $K_{r_2}, w \models B_2$.

We will need the following constructions. If Γ is the set $\{C_1, \ldots, C_n\}$ $(n \geq 0)$ then let $\&\Gamma$ be the formula $C_1 \& \cdots \& C_n$ (or $\top$ if Γ is empty). If Δ is the multiset $\{B_1, \ldots, B_n\}$ $(n \geq 0)$ then let $\otimes\Delta$ be the formula $B_1 \otimes \cdots \otimes B_n$ (or $\mathbf{1}$ if Δ is empty). Then we say an $\mathcal{L}$-style sequent $\Gamma; \Delta \longrightarrow B$ is *valid* in the model $\mathcal{M}$ if $\forall w \in \mathcal{W}, \forall r \in R$, if $K_0, w \models \&\Gamma$ and $K_r, w \models \otimes\Delta$ then $K_r, w \models B$.

Proposition 8 (Soundness Theorem) *If $\Gamma; \Delta \longrightarrow B$ has an $\mathcal{L}$-proof then $\Gamma; \Delta \longrightarrow B$ is valid in any resource indexed model.*

Proof. Let $\mathcal{M} = \{K_r \mid r \in R\}$ be a resource indexed model. The proof is by induction on the structure of an $\mathcal{L}$-proof (possibly with occurrences of the cut rules): there are two base cases and nine inductive cases (we include here the two cut-rules for $\mathcal{L}$).

identity: Given $K_0, w \models \&\Gamma$ and $K_r, w \models B$ it follows that $K_r, w \models B$. Thus, the sequent $\Gamma; B \longrightarrow B$ is valid in $\mathcal{M}$.

$\top R$: Immediate.

absorb: By the inductive hypothesis, $\Gamma, B; \Delta, B \longrightarrow C$ is provable, and hence valid in $\mathcal{M}$. Assume now that $K_0, w \models (\&\Gamma) \& B$ and $K_r, w \models \otimes\Delta$. Thus, $K_0, w \models B$ and $K_{r+0}, w \models (\otimes\Delta) \otimes B$. By the inductive hypothesis, $K_r, w \models C$ and, hence, $\Gamma, B; \Delta \longrightarrow C$ is valid in $\mathcal{M}$.

$\& R$: Immediate.

$\& L$: By the inductive hypothesis, $\Gamma; \Delta, B_i \longrightarrow C$ is valid in $\mathcal{M}$ for $i = 1$ or 2. Assume now that $K_0, w \models \&\Gamma$ and $K_r, w \models (\otimes\Delta) \otimes (B_1 \& B_2)$. Thus, there are $r_1, r_2 \in R$ such that $r = r_1 + r_2$ and $K_{r_1}, w \models \otimes\Delta$ and $K_{r_2}, w \models B_1 \& B_2$. But then $K_{r_2}, w \models B_i$ and $K_r, w \models (\otimes\Delta) \otimes B_i$. By the inductive hypothesis, $K_r, w \models C$ and, hence, $\Gamma; \Delta, B_1 \& B_2 \longrightarrow C$ is valid in $\mathcal{M}$.

$\multimap R$: By the inductive hypothesis, $\Gamma; \Delta, B \longrightarrow C$ is valid in $\mathcal{M}$. Assume now that $K_0, w \models \&\Gamma$ and $K_r, w \models \otimes\Delta$. We need to show that $K_r, w \models B \multimap C$. Assume that $w \leq w'$ and $r' \in R$ and that $K_{r'}, w' \models B$. By Lemma 7 and the definition of the satisfiability of $\otimes$, we have $K_0, w' \models \&\Gamma$ and $K_{r+r'}, w' \models (\otimes\Delta) \otimes B$. By the inductive hypothesis, $K_{r+r'}, w' \models C$. Thus we have shown that $K_r, w \models B \multimap C$ and also that the sequent $\Gamma; \Delta \longrightarrow B \multimap C$ is valid.

$\multimap L$: By the inductive hypothesis, both $\Gamma; \Delta_1 \longrightarrow B$ and $\Gamma; \Delta_2, C \longrightarrow E$ are valid in $\mathcal{M}$. Assume that $K_0, w \models \&\Gamma$ and $K_r, w \models (\otimes\Delta_1) \otimes (\otimes\Delta_2) \otimes (B \multimap C)$. Thus there are $r_1, r_2, r_3 \in R$ such that $r = r_1 + r_2 + r_3$ and $K_{r_1}, w \models \otimes\Delta_1$, $K_{r_2}, w \models \otimes\Delta_2$, and $K_{r_3}, w \models B \multimap C$. By the validity of the first sequent, $K_{r_1}, w \models B$. By the definition of the satisfiability of $\multimap$, we know $K_{r_1+r_3}, w \models C$. By the validity of the second sequent, $K_{r_1+r_2+r_3}, w \models E$. Thus, we have shown that $K_r, w \models E$ and that $\Gamma; \Delta_1, \Delta_2, B \multimap C \longrightarrow E$ is valid in $\mathcal{M}$.

$\Rightarrow R$: By the inductive hypothesis, $\Gamma, B; \Delta \longrightarrow C$ is valid in $\mathcal{M}$. Assume now that $K_0, w \models \&\Gamma$ and $K_r, w \models \otimes\Delta$. We need to show that $K_r, w \models B \Rightarrow C$. Assume that $w \leq w'$ and that $K_0, w' \models B$. By Lemma 7 and the definition of the satisfiability of &, we have $K_0, w' \models (\&\Gamma) \,\&\, B$ and $K_r, w' \models \otimes\Delta$. By the inductive hypothesis, $K_r, w' \models C$. Thus we have shown that $K_r, w \models B \Rightarrow C$ and also that the sequent $\Gamma; \Delta \longrightarrow B \Rightarrow C$ is valid in $\mathcal{M}$.

$\Rightarrow L$: By the inductive hypothesis, both $\Gamma; \emptyset \longrightarrow B$ and $\Gamma; \Delta, C \longrightarrow E$ are valid in $\mathcal{M}$. Assume that $K_0, w \models \&\Gamma$ and $K_r, w \models (\otimes\Delta) \otimes (B \Rightarrow C)$. Thus there are $r_1, r_2 \in R$ so that $r = r_1 + r_2$ and $K_{r_1}, w \models \otimes\Delta$ and $K_{r_2}, w \models B \Rightarrow C$. By the validity of the first sequent, $K_0, w \models B$. By the definition of the satisfiability of $\Rightarrow$, we know $K_{r_2}, w \models C$. By the validity of the second sequent, $K_{r_1+r_2}, w \models E$. Thus, we have shown that $K_r, w \models E$ and that $\Gamma; \Delta, B \Rightarrow C \longrightarrow E$ is valid in $\mathcal{M}$.

cut: By the inductive hypothesis, both $\Gamma'; \Delta_1 \longrightarrow B$ and $\Gamma; \Delta_2, B \longrightarrow C$ are valid in $\mathcal{M}$. Assume that $K_0, w \models \&\Gamma'$ and $K_r, w \models (\otimes\Delta_1) \otimes (\otimes\Delta_2)$. Thus $K_0, w \models \&\Gamma$ and there are $r_1, r_2 \in R$ so that $r = r_1 + r_2$ and $K_{r_1}, w \models \otimes\Delta_1$ and $K_{r_2}, w \models \otimes\Delta_2$. By the validity of the first sequent, $K_{r_1}, w \models B$ and by the validity of the second sequent, $K_{r_1+r_2}, w \models C$. Thus, we have shown that $K_r, w \models C$ and that $\Gamma'; \Delta_1, \Delta_2 \longrightarrow C$ is valid in $\mathcal{M}$.

cut!: By the inductive hypothesis, both $\Gamma'; \emptyset \longrightarrow B$ and $\Gamma, B; \Delta \longrightarrow C$ are valid in $\mathcal{M}$. Assume that $K_0, w \models \&\Gamma'$ and $K_r, w \models \otimes\Delta$. By the validity of the first sequent, $K_0, w \models B$ and by the validity of the second sequent, $K_r, w \models C$. Thus $\Gamma'; \Delta \longrightarrow C$ is valid in $\mathcal{M}$. ∎

The following proposition describes how the canonical model for our logic programming language is built and shows that it has the desired properties.

Proposition 9 (Canonical Model Theorem) *Let $\mathcal{W}$ be the set of all sets of formulas (over $\top$, &, $\multimap$, and $\Rightarrow$) and let $\leq$ be set inclusion. Let R be the set of all multisets of such formulas and let $+$ be multiset union and 0 be the empty multiset. Define $\mathcal{M} = \{K_r \mid r \in R\}$ by*

$$K_r(w) = \{A \mid A \text{ is atomic and } w; r \vdash_{\mathcal{L}} A\}.$$

Then the equivalence $\forall w \in \mathcal{W}\, \forall r \in R (w; r \vdash_{\mathcal{L}} B$ if and only if $K_r, w \models B)$ holds if and only if the two cut rules are admissible in $\mathcal{L}$.

Proof. First, assume that the two cut rules are admissible in $\mathcal{L}$. We proceed by induction on the structure of the formula B. In the base cases where B is atomic or $\top$, the conclusion is immediate. Similarly, in the inductive case where B is $B_1 \,\&\, B_2$, the conclusion also follows immediately. This leaves us with only the following two inductive cases to consider.

$B = B_1 \multimap B_2$: Assume that $w; r \longrightarrow B_1 \multimap B_2$ has a cut-free proof. By Proposition 4, the sequent $w; r, B_1 \longrightarrow B_2$ has a cut-free proof. To show $K_r, w \models B_1 \multimap B_2$, assume that $w \leq w'$ and $r' \in R$ and $K_{r'}, w' \models B_1$. By the inductive hypothesis, $w'; r' \longrightarrow B_1$ has a cut-free proof. Using the cut rule and the admissibility of cut, the sequent $w'; r + r' \longrightarrow B_2$ has a cut-free proof. By induction again, we have $K_{r+r'}, w' \models B_2$. Thus, $K_r, w \models B_1 \multimap B_2$. To show the converse, assume that $K_r, w \models B_1 \multimap B_2$. Since $w; B_1 \longrightarrow B_1$ has a cut-free proof, $K_{\{B_1\}}, w \models B_1$ (by induction). Thus, $K_{r+\{B_1\}}, w \models B_2$ and again by the inductive hypothesis, the sequent $w; r + \{B_1\} \longrightarrow B_2$ has a cut-free proof; thus, $w; r \longrightarrow B_1 \multimap B_2$ has a cut-free proof.

$B = B_1 \Rightarrow B_2$: Assume that $w; r \longrightarrow B_1 \Rightarrow B_2$ has a cut-free proof. By Proposition 4, the sequent $w, B_1; r \longrightarrow B_2$ has a cut-free proof. To show $K_r, w \models B_1 \Rightarrow B_2$, assume that $w \leq w'$ and $K_0, w' \models B_1$. By the inductive hypothesis, $w'; 0 \longrightarrow B_1$ has a cut-free proof. Using the cut! rule and the admissibility of cut!, the sequent $w'; r \longrightarrow B_2$ has a cut-free proof. By induction again, we have $K_r, w' \models B_2$. Thus, $K_r, w \models B_1 \Rightarrow B_2$. To show the converse, assume that $K_r, w \models B_1 \Rightarrow B_2$. Since $w, B_1; 0 \longrightarrow B_1$ has a cut-free proof, $K_0, w + \{B_1\} \models B_1$ (by induction). Thus, $K_{r+0}, w + \{B_1\} \models B_2$ and again by the inductive hypothesis, the sequent $w, B_1; r \longrightarrow B_2$ has a cut-free proof; thus, $w; r \longrightarrow B_1 \Rightarrow B_2$ has a cut-free proof.

Finally, we assume that the equivalence holds and use it to show that the two cut rules for $\mathcal{L}$ are admissible. Let Ξ be a proof of the sequent $\Gamma; \Delta \longrightarrow B$ possibly containing occurrences of the cut and cut! rules. We show that there is a cut-free proof of the sequent $\Gamma; \Delta \longrightarrow B$ by induction on the number of occurrences of cut and cut! rules. If Ξ has no such occurrences, then we are finished. Otherwise, pick an occurrence of a cut or cut! rule so that the two subproofs Ξ_1 and Ξ_2, whose endsequents are the premises of that cut-rule occurrence, are cut-free. We distinguish two cases. In both cases, $\Gamma \subseteq \Gamma'$

Case 1. The proof Ξ_1 proves $\Gamma'; \Delta_1 \longrightarrow B$ and Ξ_2 proves $\Gamma; \Delta_2, B \longrightarrow C$. Thus the conclusion of the cut rule is $\Gamma'; \Delta_1, \Delta_2 \longrightarrow C$. Thus, $\Gamma; \Delta_2 \longrightarrow B \multimap C$ has a cut-free proof. By the assumed equivalence, $K_{\Delta_1}, \Gamma' \models B$ and $K_{\Delta_2}, \Gamma \models B \multimap C$. By the definition of the satisfiability of $\multimap$, $K_{\Delta_1+\Delta_2}, \Gamma' \models C$ and by the equivalence again $\Gamma'; \Delta_1, \Delta_2 \longrightarrow C$ has a cut-free proof, say Ξ_3. Thus, if we replace the subproofs Ξ_1 and Ξ_2 and the cut rule with Ξ_3 in Ξ, we have reduced the number of cuts by one.

Case 2. The proof Ξ_1 proves $\Gamma'; \emptyset \longrightarrow B$ and Ξ_2 proves $\Gamma, B; \Delta \longrightarrow C$. Thus the conclusion of the cut! rule is $\Gamma'; \Delta \longrightarrow C$. Thus, $\Gamma; \Delta \longrightarrow B \Rightarrow C$ has a cut-free proof. By the assumed equivalence, $K_0, \Gamma' \models B$ and $K_\Delta, \Gamma \models B \Rightarrow C$. By the definition of the satisfiability of $\Rightarrow$, $K_\Delta, \Gamma' \models C$ and by the equivalence again $\Gamma'; \Delta \longrightarrow C$ has a cut-free proof, say Ξ_3. Thus, if we replace the subproofs Ξ_1 and Ξ_2 and the cut! rule with Ξ_3 in Ξ, we have reduced the number of cuts by one. ∎

The following is, of course, an immediate corollary of the canonical model theorem.

Proposition 10 (Completeness Theorem) *If a sequent $\Gamma; \Delta \longrightarrow B$ is valid in all resource-indexed models, then it has an $\mathcal{L}$-proof.*

7 A model of resource consumption

While we have shown several programming examples that demonstrate the usefulness of this logic programming language, we have said nothing about the practicality of implementing the language. Given the rather simple structure of proofs in $\mathcal{L}'$ and $\mathcal{L}''$, it is an easy matter to build a prototype interpreter for this logic programming language. For example, delaying the choice of universal instances of formulas on which to backchain using logic variables and unification as in other logic programming languages is an obvious option in this setting.

However, a serious computational issue that does not arise in traditional logic programming languages must be addressed. Nothing in the analysis made in Section 3 provides any guidance to an interpreter that is forced to divide up the multiset of bounded resource formulas in a proof context during the bottom-up application of the $\otimes$-R rule or the backchaining rule. For example, the sequent $\Gamma; \Delta \longrightarrow G_1 \otimes G_2$ is provable if and only if there is a partitioning of the multiset Δ into the two multisets Δ_1 and Δ_2 so that $\Gamma; \Delta_1 \longrightarrow G_1$ and $\Gamma; \Delta_2 \longrightarrow G_2$ are provable. So, if Δ has cardinality n there are 2^n such partitions of Δ. While it can be the case that for everyone of these partitions, the corresponding sequents are provable, it is much more likely (as in the examples of Section 5) that only very few of the partitions actually lead to proofs.

Clearly, a better strategy than trying each possible partition in sequence is needed if this is to be a usable logic programming language. Fortunately, given the restricted form of proof contexts, it is possible to view the process of proof building as one in which resource formulas get used and, if they are in the bounded part of the context, deleted. Thus, attempting to prove $\Gamma; \Delta \longrightarrow G_1 \otimes G_2$ first results in an attempt to prove, say G_1, and as formulas in Δ are used in backchaining inference rules in this proof, they are deleted. The resources deleted determine lazily the multiset Δ_1. If the search for a proof of G_1 is successful, the remaining multiset of formulas, implicitly equal to Δ_2, is then used to prove the second goal G_2. If the correct resources are left to prove G_2, then the compound goal $G_1 \otimes G_2$ will have been proved without splitting the context artificially.

Assume that R-formulas are defined as in Section 3. An *IO-context* is a list made up of R-formulas, !'ed R-formulas, or the special symbol del used to denote a place where an R formula has been deleted. The three-place proposition $I\{G\}O$ will denote the fact that it is possible to find a proof of G given the input I so that the resources in O remain. To make this informal notion precise, we need the following definitions regarding IO-contexts. The ternary relation $\mathsf{pickR}(I, O, R)$ holds if R occurs in the IO-context I, and O results from replacing that occurrence of R in I with the new constant del (this achieves deletion). The relation also holds if $!\,R$ occurs in I, and I and O are equal (!'ed formulas are not deleted). An IO-context O is a *subcontext* of I, denoted by the predicate $\mathsf{subcontext}(O, I)$ if O arises from replacing zero or more non-!'ed components of I with del.

Figure 15 provides a specification of the predicate $I\{G\}O$ for the propositional fragment of this logic that uses the clause $(*)$ of Section 3 to define R-formulas. The specification of $I\{G\}O$ and of the other predicates for the full range of R-formula syntax is given using Prolog in Figure 16. In that presentation, $I\{G\}O$ is written using the syntax `prove(I,O,G)`, $\otimes$ is written as `x`, $\multimap$ as `-o`, $\Rightarrow$ as `=>`, $\top$ as `erase`, and $!\,G$ as `bang(G)`. (Infix declarations for `x`, `-o`, `=>`, and `&` are missing from Figure 16, as are Horn clauses defining the atomic formulas of the object-logic via the `isA` predicate.)

The Prolog code given implements only the propositional part of this logic since Prolog has no natural representation of object-level quantification. If λProlog (Nadathur and Miller, 1988) were used for the specification, such quantifiers could be implemented directly using λ-abstractions. The resulting specification would be identical to the one given in Figure 16 except that two more clauses — one for proving a universal quantifier and one for backchaining over a universal quantifier — would need to be added.

In order to state the correctness of these specifications for $I\{G\}O$, we need the notion of the difference, $I - O$, of two IO-contexts, whenever it is the case that $\mathsf{subcontext}(O, I)$ holds: $I - O$ is the pair $\langle \Gamma, \Delta \rangle$ where Γ is the set of all formulas R such that $!\,R$ is an element of the list I (and hence O), and Δ is the multiset of all formulas R which occur in I and where the corresponding position in O is the symbol del. Thus, Δ is the multiset of formulas deleted in moving from I to O.

Proposition 11 *Let I and O be IO-contexts where O is a subcontext of I. Let $I - O$ be the pair $\langle \Gamma, \Delta \rangle$ and let G be a goal formula. The proposition $I\{G\}O$ is provable if and only if $\Gamma; \Delta \longrightarrow G$ has an $\mathcal{L}''$-proof.*

Proof. For simplicity, we show this proof only for the case where R-formulas are defined using the simpler form $(*)$ in Section 3. Notice that if $I - O$ is the pair $\langle \Gamma, \Delta \rangle$, then $\mathsf{pickR}(I, O, R)$ holds if and only if either Δ is $\{R\}$ or Δ is empty and $R \in \Gamma$.

$$\frac{}{I\{1\}I} \qquad \frac{\mathsf{subcontext}(O, I)}{I\{\top\}O} \qquad \frac{I\{G\}I}{I\{!G\}I}$$

$$\frac{I\{G_1\}M \qquad M\{G_2\}O}{I\{G_1 \otimes G_2\}O} \qquad \frac{I\{G_1\}O \qquad I\{G_2\}O}{I\{G_1 \,\&\, G_2\}O}$$

$$\frac{R{::}I\{G\}\mathsf{del}{::}O}{I\{R \multimap G\}O} \qquad \frac{!\,R{::}I\{G\}\,!\,R{::}O}{I\{R \Rightarrow G\}O}$$

$$\frac{\mathsf{pickR}(I, O, A)}{I\{A\}O} \qquad \frac{\mathsf{pickR}(I, M, G \multimap A) \qquad M\{G\}O}{I\{A\}O}$$

$$\frac{\mathsf{pickR}(I, O, G \Rightarrow A) \qquad O\{G\}O}{I\{A\}O}$$

Figure 15: Specification of an interpreter for the propositional language

We first show by induction on a proof using the inference rules in Figure 15 that if $I\{G\}O$ is provable then $\Gamma; \Delta \longrightarrow G$ has an $\mathcal{L}''$-proof. The two base cases of proving the goals $\mathbf{1}$ and $\top$ are trivial. The inductive cases are considered below.

$I\{!G\}I$ follows from $I\{G\}I$: Since Δ is $\emptyset$ in this case and since (by induction hypothesis) $\Gamma; \emptyset \longrightarrow G$ has an $\mathcal{L}''$-proof, so too does $\Gamma; \emptyset \longrightarrow\, !G$.

$I\{G_1 \otimes G_2\}O$ follows from $I\{G_1\}M$ and $M\{G_2\}O$: Let $I - M$ be $\langle\Gamma, \Delta_1\rangle$ and let $M - O$ be $\langle\Gamma, \Delta_2\rangle$. Thus, $I - O$ is $\langle\Gamma, \Delta_1 \uplus \Delta_2\rangle$. By the inductive hypothesis, $\Gamma; \Delta_1 \longrightarrow G_1$ and $\Gamma; \Delta_2 \longrightarrow G_2$ have $\mathcal{L}''$-proofs. Thus, so too does $\Gamma; \Delta_1, \Delta_2 \longrightarrow G_1 \otimes G_2$.

$I\{G_1 \,\&\, G_2\}O$ follows from $I\{G_1\}O$ and $I\{G_1\}O$: Let $I - O$ be $\langle\Gamma, \Delta\rangle$. By the inductive hypothesis, $\Gamma; \Delta \longrightarrow G_1$ and $\Gamma; \Delta \longrightarrow G_2$ have $\mathcal{L}''$-proofs. Thus, so too does $\Gamma; \Delta \longrightarrow G_1 \,\&\, G_2$.

$I\{R \multimap G\}O$ follows from $R :: I\{G\}\mathsf{del} :: O$: Let $I - O$ be $\langle\Gamma, \Delta\rangle$. Since, by induction hypothesis, $\Gamma; \Delta, R \longrightarrow G$ has an $\mathcal{L}''$-proof, so too does $\Gamma; \Delta \longrightarrow R \multimap G$.

$I\{R \Rightarrow G\}O$ follows from $!\,R :: I\{G\}\,!\,R :: O$: Let $I - O$ be $\langle\Gamma, \Delta\rangle$. Since, by induction hypothesis, $\Gamma, R; \Delta \longrightarrow G$ has an $\mathcal{L}''$-proof, so too does $\Gamma; \Delta \longrightarrow R \Rightarrow G$.

The three remaining cases involve the pickR predicate. Assume that $\mathsf{pickR}(I, O, R)$ holds and that $I - O$ is the pair $\langle\Gamma, \Delta\rangle$.

$I\{A\}O$ follows from $\mathsf{pickR}(I, O, A)$: If Δ is $\{A\}$ then $\Gamma; \Delta \longrightarrow A$ is proved by the degenerate form of the *BC* rule with no premises. Otherwise, Δ is empty, $I = O$, and $!A$ occurs in I. Then that same sequent is proved using the *BC* rule as above followed by the *absorb* rule.

$I\{A\}O$ follows from $\mathsf{pickR}(I, M, G \multimap A)$ and $M\{G\}O$: By the inductive hypothesis, $\Gamma; \Delta' \longrightarrow G$ has an $\mathcal{L}''$-proof, Ξ, where $M - O$ is $\langle\Gamma, \Delta'\rangle$. If Δ is $\{G \multimap A\}$ then $\Gamma; \Delta, \Delta' \longrightarrow A$ has an $\mathcal{L}''$-proof built from Ξ using the *BC* rule. Otherwise, Δ is empty and the *absorb* inference rule must also be used.

$I\{A\}O$ follows from $\mathsf{pickR}(I, O, G \Rightarrow A)$ and $O\{G\}O$: By the inductive hypothesis, $\Gamma; \emptyset \longrightarrow G$ has an $\mathcal{L}''$-proof, Ξ, where $O - O$ is $\langle\Gamma, \emptyset\rangle$. If Δ is $\{G \Rightarrow A\}$ then $\Gamma; \Delta \longrightarrow A$ has an $\mathcal{L}''$-proof built from Ξ using the *BC* inference rule. Otherwise, Δ is empty and the *absorb* inference rule must also be used.

The reverse implication of this proposition follows by simply reversing the constructions described above. ∎

Consider the behavior of a Prolog interpreter attempting to prove $I\{G_1 \otimes G_2\}O$. First the interpreter tries to prove $I\{G_1\}M$, for some IO-context M. If this succeeds, then $M\{G_2\}O$ is attempted. If this second attempt fails, the interpreter retries $I\{G_1\}M$ looking for some different pattern of consumption to find, hopefully, a new value for M, before re-attempting a proof of $M\{G_2\}O$. By using unification to delay the choice of the value used for the context M, that choice is made entirely lazily.

8 Related Work

There are many ways in which linear logic can be fruitfully exploited to address aspects of logic programming. Girard suggested how linear logic can be used to model the difference between the classical, "external" logic of Horn clauses and the "internal" logic of Prolog that arises from the use of depth-first search (Girard, 1987b). His suggestions were worked out in detail by Cerrito (Cerrito, 1992) to provide a logical specification of Prolog evaluation for propositional Horn clauses. Cerrito has also used classical linear logic to provide a formalization of the Clark completion theory that is sound and complete for SLDNF on allowed logic programs (Cerrito, 1990). Gabbay has used the ideas that arise from linear logic to isolate different computational aspects of proof search in a variety of systems in an attempt to identify complete yet efficient search techniques (Gabbay, 1991). In particular, he was interested in those languages in which search may be restricted to using formulas linearly. This work is conceptually related to, though somewhat broader than, the recent work in linearizing intuitionistic implication by Lincoln, Scedrov, and Shankar (Lincoln et al., 1991).

Besides our work described here, at least three proposals for new logic programming languages have been made recently that use linear logic (or, more precisely, substructural logics) for their foundation.

Andreoli and Pareschi have extended Horn clauses so that programs in the resulting language make use of the multiple conclusion nature of full linear logic (Andreoli, 1992; Andreoli and Pareschi, 1991). In the logic programming language that results, the availability of context on the right-hand side of a sequent combined with the use of the 'par' operator in the heads of clauses makes it possible to naturally support various aspects of concurrent and object-oriented programming.

Harland and Pym have proposed a fragment of linear logic as a logic programming language (Harland and Pym, 1991). As was done here, their fragment is chosen so that uniform proofs remain complete. Since having !'s in right-hand side of sequents stops several inference rule permutations from holding, their proposal disallows such right-hand sides. Thus, goal formulas are weaker than those presented here, but contexts are richer. The loss of !'ed goals, however, means that several of the examples in this paper cannot be coded directly. The authors also describe a multiple-conclusion variant of their system which captures some of the features of Andreoli and Pareschi's system.

Finally, Bollen has proposed a system which extends Prolog by adding relevant implication (Bollen, 1991). In his system, assumptions added using an embedded implication can be duplicated but cannot be discarded. The main application discussed is to expert systems. As with this work, Bollen proposes a restriction of Gentzen systems to one in which goal-directed provability is complete. His restriction is based on the notion of invertibility, rather than permutability, however, which leads to a somewhat more restricted logical language than the one presented here. In particular, arbitrary nesting of quantifiers with implications is disallowed. In the end, Bollen's

```
isR(erase).
isR(B)        :- isA(B).
isR(B1 & B2)  :- isR(B1), isR(B2).
isR(B1 -o B2) :- isG(B1), isR(B2).
isR(B1 => B2) :- isG(B1), isR(B2).

isG(1).
isG(erase).
isG(B)        :- isA(B).
isG(B1 -o B2) :- isR(B1), isG(B2).
isG(B1 => B2) :- isR(B1), isG(B2).
isG(B1 &  B2) :- isG(B1), isG(B2).
isG(B1 x  B2) :- isG(B1), isG(B2).
isG(bang(B))  :- isG(B).

prove(I,I, 1).
prove(I,O, erase)   :- subcontext(O,I).
prove(I,O, G1 & G2) :- prove(I,O,G1), prove(I,O,G2).
prove(I,O, R -o G)  :- prove(R :: I, del :: O,G).
prove(I,O, R => G)  :- prove(bang(R) :: I, bang(R) :: O,G).
prove(I,O, G1 x G2) :- prove(I,M,G1), prove(M,O,G2).
prove(I,I, bang(G)) :- prove(I,I,G).
prove(I,O, A)       :- isA(A), pickR(I,M,R), bc(M,O,A,R).

bc(I,I,A, A).
bc(I,O,A, G -o R)  :- bc(I,M,A,R), prove(M,O,G).
bc(I,O,A, G => R)  :- bc(I,O,A,R), prove(O,O,G).
bc(I,O,A, R1 & R2) :- bc(I,O,A,R1); bc(I,O,A,R2).

pickR(bang(R)::I, bang(R)::I, R).
pickR(R::I,       del::I,     R) :- isR(R).
pickR(S::I,       S::O,       R) :- pickR(I,O,R).

subcontext(del::O, R ::I) :- isR(R), subcontext(O,I).
subcontext(S::O,   S::I)  :- subcontext(O,I).
subcontext(nil,    nil).
```

Figure 16: A Prolog implementation of the IO-interpreter

system can be seen as a proper sublanguage of $\mathcal{L}$, with his relevant implication $A \rightarrow B$ being mapping into $A \multimap (A \Rightarrow B)$. The first (linear) implication assures the assumption must be used, while the second adds another copy of the assumption which can be used as many additional times as is needed. The relationship between $\mathcal{L}$ and Bollen's system is discussed in greater depth in the first author's dissertation, where an extension of $\mathcal{L}$ allowing direct representation of relevant and affine implication is also presented (Hodas, 1993).

In the area of natural language parsing, Lambek (Lambek, 1958, 1987) has used a logic that can be identified with a non-commutative variant of linear logic to infer the syntactic categories of phrases. Recently, Pereira described how the semantics of gaps could be computed using a linear logic-like context mechanism (Pereira, 1990): his approach can be formalized using the logic described here.

9 Conclusion

There have been several examples in print of the need to refine the notion of intuitionistic context found in programs written using hereditary Harrop formulas (Felty and Miller, 1988; Hodas and Miller, 1990; Pareschi and Miller, 1990; Pereira, 1990). In this paper, we proposed a refinement to hereditary Harrop formulas using a fragment of linear logic. We argued that this fragment is a sensible logic programming language by showing that interpreting it in a goal-directed fashion did not lead to incompleteness and that it has a model theory that admits canonical models. We presented an interpreter for this language that views proof construction as a process that takes in a context, deletes bounded formulas as they are used in backchaining and returns the remaining context to be consumed elsewhere. This interpreter splits contexts lazily when attempting a proof of a tensor. Finally, we presented several examples demonstrating the utility of this logic programming language.

A prototype interpreter, written in Standard ML, of the full logic programming language described here is available from the first author.

Acknowledgements

We are grateful to Jean-Marc Andreoli, Gianluigi Bellin, Jawahar Chirimar, Remo Pareschi, and Fernando Pereira for helpful conversations regarding this work. The Journal reviewers provide useful comments on an earlier draft of this paper. Both authors have been funded by ONR N00014-88-K-0633, NSF CCR-87-05596, NSF CCR-91-02753, and DARPA N00014-85-K-0018 through the University of Pennsylvania. Miller has also been supported by SERC Grant No. GR/E 78487 "The Logical Framework" and ESPRIT Basic Research Action No. 3245 "Logical Frameworks: Design Implementation and Experiment" while he was visiting the University of Edinburgh.

References

Andreoli, J.-M. (1992). Logic Programming with Focusing Proofs in Linear Logic. *Journal of Logic and Computation*, *2*(3).

Andreoli, J., and Pareschi, R. (1991). Linear Objects: Logical Processes with Built-in Inheritance. *New Generation Computing, 9:3-4.* (Special issue of papers selected from ICLP'90).

Apt, K. R., and van Emden, M. H. (1982). Contributions to the theory of logic programming. *Journal of the ACM*, *29*(3), 841 – 862.

Bollen, A. W. (1991). Relevant Logic Programming. *Journal of Automated Reasoning*, *7*(4), 563–586.

Bonner, A. J., McCarty, L. T., and Vadaparty, K. (1989). Expressing Database Queries with Intuitionistic Logic. In *Logic Programming: Proceeding of the North American Conference*, pp. 831–850.

Bowen, K. A., and Kowalski, R. A. (1982). Amalgamating Language and Metalanguage in Logic Programming. In Clark, K., and Tarnlund, S.-A. (Eds.), *Logic programming*, Vol. 16 of *APIC studies in data processing*, pp. 153 – 172. Academic Press.

Cerrito, S. (1990). A Linear Semantics for Allowed Logic Programs. In Mitchell, J. (Ed.), *Proceedings of the Fifth Annual Symposium on Logic in Computer Science, Philadelphia, PA*, pp. 219 – 227.

Cerrito, S. (1992). A Linear Axiomatization of Negation as Failure. *Journal of Logic Programming*, *12*(1 & 2), 1 – 24.

Church, A. (1940). A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, *5*, 56–68.

Dyckhoff, R. (1992). Contraction-free sequent calculi for intuitionistic logic. *Journal of Symbolic Logic*, *57*(3).

Felty, A. (1993). Implementing Tactics and Tacticals in a Higher-Order Logic Programming Language. *Journal of Automated Reasoning*, *11*(1), 43–81.

Felty, A., and Miller, D. (1988). Specifying theorem provers in a higher-order logic programming language. In *Ninth International Conference on Automated Deduction*, pp. 61 – 80 Argonne, IL. Springer-Verlag.

Gabbay, D. M., and Reyle, U. (1984). N-Prolog: An Extension of Prolog with Hypothetical Implications. I. *Journal of Logic Programming*, *1*, 319 – 355.

Gabbay, D. M. (1991). Algorithmic Proof with Diminishing Resources, Part 1. In Börger, E. (Ed.), *CSL'90. Fourth Workshop on Computer Science Logic*, No. 533 in Lecture Notes in Computer Science. Springer-Verlag.

Gentzen, G. (1969). Investigations into Logical Deductions, 1935. In Szabo, M. E. (Ed.), *The Collected Papers of Gerhard Gentzen*, pp. 68–131. North-Holland Publishing Co., Amsterdam.

Girard, J.-Y. (1987a). Linear Logic. *Theoretical Computer Science*, *50*, 1–102.

Girard, J.-Y. (1987b). Towards a Geometry of Interaction. In *Categories in Computer Science*, Vol. 92 of *Contemporary Mathematics*, pp. 69 – 108. AMS.

Girard, J.-Y. (1991). On the unity of logic. Tech. rep. 26, Université Paris VII.

Hallnäs, L., and Schroeder-Heister, P. (1990). A Proof-Theoretic Approach to Logic Programming. 1. Clauses as rules. *Journal of Logic and Computation*, 261–283.

Harland, J., and Pym, D. (1991). The Uniform Proof-Theoretic Foundation of Linear Logic Programming (Extended Abstract). In Saraswat, V., and Ueda, K. (Eds.), *Proceedings of the 1991 International Logic Programming Symposium, San Diego*, pp. 304 – 318 San Diego. MIT Press.

Hodas, J. (1992). Specifying Filler-Gap Dependency Parsers in a Linear-Logic Programming Language. In Apt, K. (Ed.), *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pp. 622 – 636.

Hodas, J. (1993). *Logic Programming in Intuitionistic Linear Logic: Theory, Design, and Implementation*. Ph.D. thesis, University of Pennsylvania, Department of Computer and Information Science.

Hodas, J., and Miller, D. (1990). Representing Objects in a Logic Programming Language with Scoping Constructs. In Warren, D. H. D., and Szeredi, P. (Eds.), *1990 International Conference in Logic Programming*, pp. 511 – 526. MIT Press.

Hodas, J., and Miller, D. (1991). Logic Programming in a Fragment of Intuitionistic Linear Logic: Extended Abstract. In Kahn, G. (Ed.), *Sixth Annual Symposium on Logic in Computer Science*, pp. 32 – 42 Amsterdam.

Lambek, J. (1958). The mathematics of sentence structure. *American Mathematical Monthly*, *65*, 154 – 169.

Lambek, J. (1987). Multicategories Revisited. In *Categories in Computer Science*, Vol. 92 of *Contemporary Mathematics*, pp. 217 – 239. AMS.

Lincoln, P., Scedrov, A., and Shankar, N. (1991). Linearizing Intuitionistic Implication. In Kahn, G. (Ed.), *Sixth Annual Symposium on Logic in Computer Science*, pp. 51–62.

McCarty, L. T. (1988). Clausal Intuitionistic Logic I. Fixed Point Semantics. *Journal of Logic Programming*, *5*, 1 – 31.

Miller, D. (1989a). Lexical Scoping as Universal Quantification. In *Sixth International Logic Programming Conference*, pp. 268–283 Lisbon, Portugal. MIT Press.

Miller, D. (1989b). A logical analysis of modules in logic programming. *Journal of Logic Programming*, *6*, 79 – 108.

Miller, D. (1990). Abstractions in logic programming. In Odifreddi, P. (Ed.), *Logic and Computer Science*, pp. 329 – 359. Academic Press.

Miller, D. (1992). Abstract Syntax and Logic Programming. In *Logic Programming: Proceedings of the First and Second Russian Conferences on Logic Programming*, No. 592 in Lecture Notes in Artificial Intelligence, pp. 322–337. Springer-Verlag. Also available as technical report MS-CIS-91-72, UPenn.

Miller, D., Nadathur, G., Pfenning, F., and Scedrov, A. (1991). Uniform Proofs as a Foundation for Logic Programming. *Annals of Pure and Applied Logic*, *51*, 125–157.

Nadathur, G., and Miller, D. (1988). An Overview of λProlog. In *Fifth International Logic Programming Conference*, pp. 810–827 Seattle, Washington. MIT Press.

Nadathur, G., and Miller, D. (1990). Higher-order Horn Clauses. *Journal of the ACM*, *37*(4), 777 – 814.

Pareschi, R. (1989). *Type-driven Natural Language Analysis*. Ph.D. thesis, University of Edinburgh.

Pareschi, R., and Miller, D. (1990). Extending Definite Clause Grammars with Scoping Constructs. In Warren, D. H. D., and Szeredi, P. (Eds.), *1990 International Conference in Logic Programming*, pp. 373–389. MIT Press.

Paulson, L. C. (1990). Isabelle: The Next 700 Theorem Provers. In Odifreddi, P. (Ed.), *Logic and Computer Science*, pp. 361 – 386. Academic Press.

Pereira, F. C. N. (1990). Semantic Interpretation as Higher-Order Deduction. In *Proceedings of the Second European Workshop on Logics and AI*. Springer-Verlag.

Pereira, F. C. N., and Shieber, S. M. (1987). *Prolog and Natural-Language Analysis*, Vol. 10. CLSI, Stanford, CA.

Pereira, F. C. N., and Warren, D. H. D. (1980). Definite Clauses for Language Analysis. *Artificial Intelligence*, *13*, 231 – 278.

Pfenning, F. (1988). Partial Polymorphic Type Inference and Higher-Order Unification. In *Proceedings of the ACM Lisp and Functional Programming Conference.*