April 1991

Logic Programming in a Fragment of Intuitionistic Linear Logic: Extended Abstract

Joshua S. Hodas
University of Pennsylvania

Dale Miller
University of Pennsylvania

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-91-32.

Logic Programming in a Fragment of Intuitionistic Linear Logic: Extended Abstract

Abstract

Logic programming languages based on fragments of intuitionistic logic have recently been developed and studied by several researchers. In such languages, implications are permitted in goals and in the bodies of clauses. Attempting to prove a goal of the form $D \supset G$ in a context τ leads to an attempt to prove the goal G in the extended context $\tau \cup \{D\}$. While an intuitionistic notion of context has many uses, it has turned out to be either too powerful or too limiting in several settings. We refine the intuitionistic notion of context by using a fragment of Girard's linear logic that includes additive and multiplicative conjunction, linear implication, universal quantification, the "of course" exponential, and the constants 1 (the empty context) and T (for "erasing" contexts). After presenting our fragment of linear logic, which contains the hereditary Harrop formulas, we show that the logic has a goal-directed interpretation. We also show that the non-determinism that results from the need to split contexts in order to prove a multiplicative conjunction can be handled by viewing proof search as a process that takes a context, consumes part of it, and returns the rest (to be consumed elsewhere). The complete specification of an interpreter for this logic is presented. 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.

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-91-32.

Logic Programming In A Fragment Of Intuitionistic Linear Logic: Extended Abstract

MS-CIS-91-32
LOGIC & COMPUTATION 32

Joshua S. Hodas
Dale Miller

Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania
Philadelphia, PA 19104-6389

April 1991

Logic Programming in a Fragment of Intuitionistic Linear Logic: Extended Abstract*

Joshua S. Hodas
Computer Science Dept.
University of Pennsylvania
Philadelphia, PA 19104-6389 USA
hodas@saul.cis.upenn.edu

Dale Miller
LFCS, Computer Science Dept.
University of Edinburgh, KB
Edinburgh EH9 3JZ Scotland
dmi@lfcs.ed.ac.uk

Abstract

Logic programming languages based on fragments of intuitionistic logic have recently been developed and studied by several researchers. In such languages, implications are permitted in goals and in the bodies of clauses. Attempting to prove a goal of the form $D \supset G$ in a context Γ leads to an attempt to prove the goal G in the extended context $\Gamma \cup \{D\}$. While an intuitionistic notion of context has many uses, it has turned out to be either too powerful or too limiting in several settings. We refine the intuitionistic notion of context by using a fragment of Girard's linear logic that includes additive and multiplicative conjunction, linear implication, universal quantification, the "of course" exponential, and the constants 1 (the empty context) and $\top$ (for "erasing" contexts). After presenting our fragment of linear logic, we show that the logic has a goal-directed interpretation. We also show that the non-determinism that results from the need to split contexts in order to prove a multiplicative conjunction can be handled by viewing proof search as a process that takes a context, consumes part of it, and returns the rest (to be consumed elsewhere). 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.

*This paper appears in LICS91. A typographical error in Figure 5 has been fixed. Both authors have been funded by ONR N00014-88-K-0633, NSF CCR-87-05596, and DARPA N00014-85-K-0018 through the University of Pennsylvania from which Miller is on a one year leave. 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."

1 Introduction

Fragments of intuitionistic first-order and higher-order logics have been used for several years 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 existentials) have been used as both specification languages for proof systems [7, 8, 25, 29] and as the basis of logic programming languages [9, 12, 18, 20, 21]. Much of the expressivity of such systems derives from the the intuitionistic proof rule which states that to prove an implicational goal of the form $D \supset G$ in the presence of a proof context Γ, the context is augmented with D and a proof of G is attempted 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 presence of changing antecedents (contexts) within intuitionistic sequent proofs can be exploited in many ways. In theorem provers, it can be used to store the current assumptions and eigen-variables of a proof; in natural language parsers, it can be used to store assumed gaps when parsing relative clauses; in database programs, it can be used to store the state of the data base; in logic programs, it 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 other applications they are too limiting. Speaking operationally, one problem that appears frequently is that once an item is placed into a context, it is not possible to remove it, short of stopping the process that created the context. In a logical sense, a formula in a context is available to be used any number of

times. Thus, the freely available contraction rule for intuitionistic contexts means that 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 [8, 14]. With such an approach, however, it is not possible to specify any variations to the contraction rule: 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 [24]. 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" [27]).

- 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 are not straightforward [3, 9, 19].

- A notion of state encapsulation can be approximated using intuitionistic logic [15]. An object's state can be represented by assumptions in a context. Updating that state, however, means changing those assumptions. The only change allowed with intuitionistic contexts is that of augmenting the state's representative assumptions. Thus, an object's state becomes progressively more nondeterministic: seldom the correct notion of state.

Each of these problems can, however, be addressed by adopting a more refined notion of context. In this extended abstract, we examine a fragment of linear logic that makes a suitable logic programming language and permits very natural formulations that address all of the above problems.

2 Language Design Issues

Consider a first-order logic based on the logical constants $\otimes$ (tensor), & (additive conjunction), $\multimap$ (linear implication), ! (of-course), $\forall$, 1, $\top$. Our presentation of sequent proof systems for this logic will differ from the usual one [1, 10] in some simple ways. First, the antecedent (left-hand side) of a sequent arrow will always contain a single formula: the comma traditionally used in the antecedent of a sequent will be formally identified with the connective $\otimes$, and an empty antecedent with the formula 1. Thus no introduction rule for $\otimes$ on the left is needed. Second, antecedents (contexts) are identified if they can be shown to be equal via the equations for associativity, commutativity, identity, and the idempotency of !'ed formulas. That is, two contexts are considered equal for the purposes of building sequent proofs if one arises from the other by applying the following equations to subformulas that are either top-level or in the scope of $\otimes$'s only:

$$t \otimes s = s \otimes t, (t \otimes s) \otimes r = t \otimes (s \otimes r), t \otimes 1 = t, !t \otimes !t = !t.$$

By employing this notion of equality on contexts, the structural rules of interchange, contraction for !'ed formulas, and thinning for the formula 1 (the empty context) are used implicitly. If s is a subformula of t that occurs in t in the scope of only $\otimes$'s and is neither a top-level $\otimes$ nor equal to 1, then s is a *component* of t. If the outermost connective in t is not a $\otimes$, then t has only one component.

A cut-free, sequent style proof system $\mathcal{L}$ for this fragment of linear logic is given in Figure 1. The syntactic variables Γ, B, C denote formulas. The syntactic variable $!\Gamma$ denotes a *thinable* formula, that is, a formula all of whose components are either 1 or $!B$ (for some B). The standard eigen-variable condition is assumed for the $\forall R$ rule. A proof rule that says that if Γ and Γ' are equal contexts and if $\Gamma \longrightarrow C$ has a proof then $\Gamma' \longrightarrow C$ has a proof is assumed to be used whenever it is needed.

Proposition 1 *Let B be a first-order formula over the logical constants $\otimes, \&, \multimap, !, \forall, 1, \top$. The sequent $1 \longrightarrow B$ is provable in $\mathcal{L}$ if and only if B is provable in linear logic. For this sublogic, provability in classical and intuitionistic linear logic coincide [30].*

Two formulas B and C are *equivalent*, written $B \equiv C$, if the sequents $B \longrightarrow C$ and $C \longrightarrow B$ are provable in $\mathcal{L}$. We note the following equivalences:

$$1 \equiv !\top, \quad !(B \,\&\, C) \equiv !B \otimes !C,$$
$$B \multimap (C \,\&\, D) \equiv (B \multimap C) \,\&\, (B \multimap D),$$
$$B \multimap (\forall x.C) \equiv \forall x(B \multimap C) \text{ (provided } x \text{ not free in } B)$$
$$\forall x(B \,\&\, C) \equiv (\forall x.B) \,\&\, (\forall x.C)$$

$$\frac{}{!\Gamma \otimes B \longrightarrow B}\ \text{identity} \qquad \frac{}{!\Gamma \longrightarrow 1}\ 1R \qquad \frac{}{\Gamma \longrightarrow \top}\ \top R$$

$$\frac{\Gamma \longrightarrow B \qquad \Gamma \longrightarrow C}{\Gamma \longrightarrow B \,\&\, C}\ \&R$$

$$\frac{\Gamma \otimes B_i \longrightarrow C}{\Gamma \otimes (B_1 \,\&\, B_2) \longrightarrow C}\ \&L\ (i = 1, 2)$$

$$\frac{\Gamma \otimes B \longrightarrow C}{\Gamma \longrightarrow B \multimap C}\ \multimap R \qquad \frac{\Gamma \longrightarrow B \qquad \Delta \otimes C \longrightarrow E}{\Gamma \otimes (B \multimap C) \otimes \Delta \longrightarrow E}\ \multimap L$$

$$\frac{\Gamma \longrightarrow B \qquad \Delta \longrightarrow C}{\Gamma \otimes \Delta \longrightarrow B \otimes C}\ \otimes R$$

$$\frac{\Gamma \otimes B \otimes !B \longrightarrow C}{\Gamma \otimes !B \longrightarrow C}\ !L \qquad \frac{!\Gamma \longrightarrow B}{!\Gamma \longrightarrow !B}\ !R$$

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

Figure 1: $\mathcal{L}$: A Proof System for a Fragment of Linear Logic

Given the first equivalence, the constant 1 could be eliminated from our consideration, although there seems to be no good reason to do so.

It is possible to strengthen $\mathcal{L}$ proofs a bit further, by observing that the multiplicative inference rules, $\multimap L$ and $\otimes R$, can be assumed to behave *additively* with respect of !'ed formulas. That is, we can assume that, for both of these inference rules, if the antecedent of the conclusion has the component $!B$, then $!B$ is a component of the antecedent of both premises. A proof where this additional restriction on $\multimap L$ and $\otimes R$ holds will be called a *!-additive* proof. Thus, a !-additive proof of the sequent $!B \otimes \Gamma \longrightarrow C$ is such that the antecedent of all sequents in the proof contain a top-level occurrence of $!B$.

Proposition 2 *The sequent $1 \longrightarrow B$ has a !-additive $\mathcal{L}$ proof if and only if B is provable in linear logic.*

The logic considered so far does not yield a satisfactory logic programming language because it cannot be interpreted in a completely *goal-directed* fashion. Following [19, 21], we formalize the notion of goal-directed provability by the simple notion of a *uniform proof*: a cut-free, sequent proof is uniform if whenever a sequent in the proof has a non-atomic succedent (right-hand side), that sequent is the conclusion of a right introduction rule. A logical system is an *(abstract) logic programming language* if restricting to uniform proofs preserves completeness. Thus, if a sequent $\Gamma \longrightarrow B$ is provable in a logic programming language, it is always possible to find a proof by first concentrating only on the succedent of the sequent, breaking down its logical structure and reflecting it into the proof. Left introduction rules only need to be considered when atomic succedents are encountered.

Restricting provability in $\mathcal{L}$ to uniform proofs does not maintain completeness, however. For example, the sequents $!a \,\&\, b \longrightarrow !a$ and $b \otimes (b \multimap !a) \longrightarrow !a$ are provable in $\mathcal{L}$ but do not have uniform proofs. Given this observation there seem to be two approaches to take: restrict the language so that a !'ed formula, for example, never appears in a succedent, or restrict formulas that occur in contexts so that the above counterexamples (and their kin) do not appear. The first approach has been taken in the recent paper [13]. We shall take the second approach, however, for three reasons. First, some of the examples mentioned in Section 1 are most successfully explained using ! in succedents. Second, it can be seen that if the intuitionistic connectives of hereditary Harrop formulas are mapped into formulas using linear connectives [10], occurrences of ! never appear immediately under & or as the conclusion of $\multimap$. Maintaining such a restriction rules out the above two examples sequents and would seem a sensible approach since the logic here is a refinement of hereditary Harrop formulas. Third, it will be useful for an interpreter to be able to examine a component of its context and to determine immediately whether it is a "use once" formula or a "use any number" formula. If ! were permitted as a strictly positive subformula occurrence of a & or a $\multimap$, it would not be possible to make this distinction immediately. In the first example, there is a non-deterministic choice to be made before this distinction can be made with the formula $!a \,\&\, b$, and in the second a subproof needs to be performed (to prove b) before it can be seen that the formula $b \multimap !a$ yields a !'ed formula. Our restrictions on the logic will be those that exclude these kinds of formulas in contexts.

Below is the definition of three classes of formulas: R for resource formulas, D for definite formulas (program formulas), and G for goal formulas (queries).

$$R := \top \mid A \mid R_1 \,\&\, R_2 \mid G \multimap R \mid \forall x.R$$
$$D := 1 \mid R \mid !R \mid D_1 \otimes D_2$$
$$G := 1 \mid \top \mid A \mid G_1 \,\&\, G_2 \mid D \multimap G \mid G_1 \otimes G_2 \mid !G \mid \forall x.G$$

It is possible to show that if $\mathcal{L}$ proofs are restricted to proofs of sequents of the form $D \longrightarrow G$ then uniform proofs are complete. The restricted logic is, therefore, an abstract logic programming language. This result can be strengthened by first making the following definitions.

Let R be a closed resource formula. Let $|R|$ be the smallest set of pairs $\langle \mathcal{G}, R' \rangle$, where $\mathcal{G}$ is a multiset of goal formulas and R' is a resource formula, such that

1. $\langle \emptyset, R \rangle \in |R|$
2. if $\langle \mathcal{G}, R_1 \,\&\, R_2 \rangle \in |R|$ then both $\langle \mathcal{G}, R_1 \rangle \in |R|$ and $\langle \mathcal{G}, R_2 \rangle \in |R|$
3. if $\langle \mathcal{G}, \forall x.R' \rangle \in |R|$ then for all closed terms t, $\langle \mathcal{G}, R'[t/x] \rangle \in |R|$
4. if $\langle \mathcal{G}, G \multimap R' \rangle \in |R|$ then $\langle \mathcal{G} \cup \{G\}, R' \rangle \in |R|$ (here, $\mathcal{G} \cup \{G\}$ is multiset union)

Informally, if $\langle \mathcal{G}, A \rangle \in |R|$ then the formula R can be used to establish the atomic formula A if each of the goal formulas in $\mathcal{G}$ can be established; that is, A might be proved by backchaining over R. Let $\mathcal{L}'$ be the proof system that results from replacing the identity, $\multimap L$, $\&L$, and $\forall L$ rules in Figure 1 with the *backchaining* inference rule in Figure 2. Thus, there are only two left rules in $\mathcal{L}'$, namely, backchaining and $!L$. The notion of !-additive proofs is easily extended to $\mathcal{L}'$.

The following proposition helps to justify our claim that our restriction to linear formulas yields a logic programming language.

Proposition 3 *Let G be a goal formula. The sequent $1 \longrightarrow G$ has a !-additive, uniform proof in $\mathcal{L}'$ if and only if G is provable in linear logic.*

This proposition can be proved by showing that any proof in $\mathcal{L}$ can be transformed directly into a uniform proof in $\mathcal{L}'$ by a series of permutations of inference rules. Similar permutation arguments can be found in [19, 21].

Since the formulas allowed in antecedents are different from those allowed in succedents, a cut rule cannot be stated for this logic programming language. The collection of formulas in the intersection between D- and G-formulas is, however, non-empty and can be described using the following two classes of formulas:

$$\frac{\Gamma_1 \longrightarrow G_1 \quad \ldots \quad \Gamma_n \longrightarrow G_n}{\Gamma_1 \otimes \cdots \otimes \Gamma_n \otimes !\Gamma \otimes R \longrightarrow A}\; BC$$

Figure 2: Backchaining: provided $n \geq 0$, A is atomic, and $\langle \{G_1, \ldots, G_n\}, A \rangle \in |R|$.

$$S := \top \mid A \mid S_1 \,\&\, S_2 \mid M \multimap S \mid \forall x.S$$
$$M := 1 \mid S \mid !S \mid M_1 \otimes M_2.$$

The collection of M formulas is precisely those formulas which are both goals and definite formulas. A cut-elimination theorem can be stated and proved for a restriction to M formulas only. Modulo the equivalences $1 \multimap S \equiv S$ and $(M_1 \otimes M_2) \multimap S \equiv M_1 \multimap M_2 \multimap S$, the collection S can also be described as simply that collection of formulas freely generated from &, $\forall$, and both linear and intuitionistic implications: that is, S can be defined by

$$S := \top \mid A \mid S_1 \,\&\, S_2 \mid S_1 \multimap S_2 \mid !S_1 \multimap S_2 \mid \forall x.S.$$

It is worth noting some simple variants of the definitions of R, D, and G formulas. First, it is an easy matter to add the existential quantifier and the additive disjunction $\oplus$ to goal formulas: all the results below generalize easily to this case. (It is not possible to add existential quantifiers or $\oplus$ to either resource formulas or definite formulas without seriously losing the completeness of uniform proofs.) Second, given the equivalences mentioned above, it is possible to replace the definition for resources with

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

Such a simplification makes the presentation of backchaining a bit simplier, a fact used in the next section.

A final comment: If the uniformity conditions are applied directly to proofs in Girard's original formulation of intuitionistic linear logic [10] then completeness is lost quickly, even for the restricted language described above. For example, the sequents $a \otimes b \longrightarrow b \otimes a$ and $!a \longrightarrow !a \otimes !a$ have proofs but not uniform proofs. In the first case, an $\otimes$L rule is required to convert the antecedent to a, b and in the second case, a contraction is needed to form the antecedent $!a, !a$ (reading proofs bottom-up). Since these left rules must be done before any right rule, these sequents do not have uniform proofs. In this setting, however, such a failure is easy to circumvent: in both cases, the contexts $a \otimes b$ and a, b, and $!a$ and $!a, !a$, do not seem different in any significant computational sense. Formally identifying

them as is done in $\mathcal{L}$ and $\mathcal{L}'$ produces proof systems in which uniform proofs cover a greater number of cases.

3 An Input/Output Model of Resource Consumption

The information in Proposition 3 is enough to form the basis of the design of a prototype interpreter for this restriction of linear logic. Such an interpreter will not, however, receive any guidance when it needs to break up an antecedent in order to apply the $\otimes R$ or backchaining rules in a bottom-up fashion. Given that proofs can be assumed to be !-additive, a theorem prover only needs to decide how to break up the non-!'ed formulas in the context. Nevertheless, trying all possible partitions of those formulas is, of course, exponential in the number of such formulas. Clearly, a better strategy is desirable.

Given our restriction on what kinds of formulas can appear in contexts (namely definite formulas), it is possible to view the process of proof building as one where resource formulas get used and, if they are not !'ed, deleted. Thus, one way to attempt a proof of $G_1 \otimes G_2$ from a D-formula D is to first try to prove G_1, deleting non-!'ed R-formulas as they are used in backchaining. If the search for a proof of G_1 is successful, the resulting context, say D', is then used to prove the second goal G_2. If the correct amount of 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.

With this motivation, we can define the predicate $I\{G\}O$, where I and O are D-formulae, to mean informally that when given input I, a proof for G can be found that returns the resources in O. To make this informal notion precise, we need the following definitions regarding definite formulas. Notice that components of a definite formula D are either 1 or of the form R or $!R$. The ternary relation $\mathsf{pickR}(I, O, R)$ holds if R is a component of I and O results from replacing that occurrence of R in I with 1 (this achieves deletion). The relation also holds if $!R$ is a component of I, and I and O are equal (!'ed formulas are not deleted). A definite formula O is a *subtensor* of I, denoted by the predicate $\mathsf{subtensor}(I, O)$ if O arises from replacing zero or more non-!'ed components of I with 1. The formula I is *thinable*, written $\mathsf{thinable}(I)$, if all of its components are either 1 or !'ed formulas.

Figure 3 provides a specification of the predicate $I\{G\}O$ for the propositional fragment of this logic in which the simpler definition of resource formulas (see (*) above) is used (no equations are needed to characterize contexts in this specification). A fuller specification of $I\{G\}O$ and the other predicates is given using Prolog in Figure 4. In that presentation, $I\{G\}O$ is written using the syntax `prove(I,O,G)`, $\otimes$ is written as `x`, $\multimap$ as `-o`, $\top$ as `true`, and $!G$ as `bang(G)`. (Infix declarations for `x`, `-o`, and `&` are missing from Figure 4, as are Horn clauses defining the atomic formulas of the object-logic via the `isA` predicate.) The Prolog code only implements the propositional part of this logic since Prolog has no natural representation of object-level quantification. If λProlog [22] is used for this specification, such quantifiers can be implemented directly using λ-abstractions. The resulting specification in λProlog would be identical to that given in Figure 4 except that two more Horn clauses — one for proving a universal quantifier and one for backchaining over a universal quantifier — would need to be added.

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

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

$$\frac{D \otimes I\{G\}B \otimes O \quad \mathsf{thinable}(B)}{I\{D \multimap G\}O}$$

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

Figure 3: Specification of an interpreter for the propositional language

In order to state the correctness of these specifications of $I\{G\}O$, we need the notion of the difference, $I - O$, of two definite formulas, whenever it is the case that $\mathsf{subtensor}(I, O)$. This is defined by the following equations:

$$1 - 1 = 1, \quad R - R = 1, \quad !R - !R = !R, \quad R - 1 = R,$$
$$(I_1 \otimes I_2) - (O_1 \otimes O_2) = (I_1 - O_1) \otimes (I_2 - O_2).$$

Here, of course, R denotes a resource formula.

Proposition 4 *Let I and O be contexts (definite formulas) and let G be a goal formula. The proposition $I\{G\}O$ is provable if and only if $(I - O) \multimap G$ is provable in linear logic.*

Thus, a complete procedure for proving queries from Horn clauses can be used to prove sequents of the form $D \longrightarrow G$ in linear logic. A depth-first search procedure for Horn clauses can also provide a depth-first search procedure for linear logic. Consider the behaviour 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 definite formula 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 a new value for M. If such a value is found, then $M\{G_2\}O$ is re-attempted. Similarly, in attempting to prove $I\{D \multimap G\}O$, the interpreter will first attempt to prove $D \otimes I\{G\}B \otimes O$ and then check that B is a thinable context. If this is not the case, then the interpreter must retry $D \otimes I\{G\}B \otimes O$ looking for a new value for B.

Proposition 5 *The goal formula G is provable in linear logic if and only if the proposition $1\{G\}1$ is provable.*

As the process of constructing a proof of $1\{G\}1$ proceeds, a Horn clause interpreter would generate goals of the form

$$T_1 \otimes \cdots \otimes T_n \otimes 1\{G'\}S_1 \otimes \cdots \otimes S_n \otimes 1,$$

where $T_1, \ldots, T_n, S_1, \ldots, S_n$ are definite formulas and $n \geq 0$ is the number of $\multimap R$ rules encountered on the path from the root. Thus, contexts act as stacks, and corresponding elements, say T_i and S_i, are related by the subtensor relation.

It is worth noting that, in this setting, proving the goal $!G$ from a D-formula is equivalent to proving the goal $1 \,\&\, G$ from that same D-formula.

4 Context Management in Theorem Provers

Intuitionistic logic is a useful meta-logic for the specification of provability of various object-logics. For example, consider axiomatizing provability in propositional, intuitionistic logic (over the logical symbols **imp**, **and**, **or**, and **false** for object-level implication, conjunction, disjunction, and absurdity). A very natural axiomization of implication introduction is (ignoring the actual construction of proofs)

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

where **pv** and **hyp** are the meta-level predicates denoting provability and hypothesis. Here => denotes meta-level implies and :- denotes its converse. We also

```
isR(true).
isR(B)        :- isA(B).
isR(B1 & B2)  :- isR(B1), isR(B2).
isR(B1 -o B2) :- isG(B1), isR(B2).

isD(1).
isD(B)        :- isR(B).
isD(bang(B))  :- isR(B).
isD(B1 x B2)  :- isD(B1), isD(B2).

isG(1).
isG(true).
isG(B)        :- isA(B).
isG(B1 -o B2) :- isD(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, true)    :- subtensor(I, O).
prove(I,O, G1 & G2) :- prove(I,O,G1),
                       prove(I,O,G2).
prove(I,O, D -o G)  :- prove(D x I,B x O,G),
                       thinable(B).
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),
                       backchain(M,O,A,R).

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

pickR(T1 x T, T2 x T, R) :- pickR(T1,T2,R).
pickR(T x T1, T x T2, R) :- pickR(T1,T2,R).
pickR(R,      1,      R) :- isR(R).
pickR(bang(R),bang(R),R).

thinable(1).
thinable(bang(T)).
thinable(T x S) :- thinable(T), thinable(S).

subtensor(T1 x T2, S1 x S2) :- subtensor(T1,S1),
                               subtensor(T2,S2).
subtensor(1,       1).
subtensor(bang(R), bang(R)).
subtensor(R,       1) :- isR(R).
subtensor(R,       R) :- isR(R).
```

Figure 4: A Prolog implementation of the interpreter

```
pv(A and B) o- pv(B) & pv(A).
pv(A imp B) o- hyp(A) -o pv(B).
pv(A or  B) o- pv(A).
pv(A or  B) o- pv(B).
pv(G) o- hyp(A and B) x
         (hyp(A) x hyp(B) -o pv(G)).
pv(G) o- hyp(A or  B) x ((hyp(A) -o pv(G)) &
                         (hyp(B) -o pv(G))).
pv(G) o- hyp(C imp B) x
         (hyp(C imp B) -o pv(C)) &
         (hyp(B) -o pv(G)).
pv(G) o- hyp(false) x true.
pv(G) o- hyp(G)     x true.
```

Figure 5: A Specification of An Intuitionistic Propositional Object-Logic

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

isatom(p) o- 1.
isatom(q) o- 1.
isatom(r) o- 1.
```

Figure 6: A Contraction-Free Formulation of $\supset L$

adopt the convention that any capital letters in a formula that are not explicitly quantified are assumed to be universally quantified at the outermost scope of the displayed formula. Operationally, this formula states that one way to prove `A imp B` is to add `hyp(A)` to the context and attempt a proof of `pv(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)).
```

(where a comma denotes a meta-level conjunction). Operationally, this formula states that in order to prove some formula `G`, first check to see if there is a conjunction, say `A and B`, in the context and, if so, attempt a proof of `G` in which the context is extended with the two hypotheses `A` and `B`. Other introduction and elimination rules can be axiomatized 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 in the meta-logic (the logic programming world) if and only if there is a proof in the object-logic.

This method of specifying provability does not extend naturally to 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. Similarly, it is not possible to logically express the fact that in intuitionistic propositional logic, contraction is needed only for implication elimination and not for any other logical connectives. If we replace the intuitionistic meta-logic with our refinement using linear logic, however, these observations about contraction in intuitionistic logic can be stated. Figure 5 is such a specification. Here, and in the rest of the code segments in the paper, `o-` denotes the converse of `-o`. Formulas carry the same implicit quantifier assumption mentioned above. In addition, each quantified formula is assumed to be !'ed at the outermost level. Finally, the individual formulas are tied together with $\otimes$'s to form a single D formula for the proof context.

In the modified specification, a hypothesis is both "read from" and "written into" the context during the elimination of implications. All other elimination rules simply "read from" the context; they do not "write back." The last two formulas in Figure 5 use a $\otimes$ with $\top$: this allows for all unused hypotheses to be erased.

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 for this example, a contraction-free presentation of propositional intuitionistic logic is given in [6]. That presentation can be expressed directly in this setting by replacing the one formula specifying implication elimination in Figure 5 with the (partial) axiomatization of object-level atomic formulas and the five special cases of implication elimination in Figure 6. Executing this linear logic program in depth-first mode yields a decision procedure for propositional intuitionistic logic.

5 The Modality of !'ed Formulas

One extension to logic programming languages that has been studied for several years is the *demo*-predicate [4]. The intended meaning of attempting a query of the form $demo(D, G)$ in context (program) Γ is simply attempting the query G in the context containing only D; that is, the main context is forgotten during the scope of the *demo*-predicate. A use of a !'ed goal has a related meaning. Consider proving the sequent $!R_1 \otimes R_2 \longrightarrow !G_1 \otimes G_2$. Given our analysis of proofs in Section 2, this is provable if and only if the two sequents $!R_1 \longrightarrow G_1$ and $!R_1 \otimes R_2 \longrightarrow G_2$ are provable. In other words, the use of the "of course" operator forces G_1 to be proved in a context that contains only !'ed formulas. In a sense, since non-!'ed resources can come and go within contexts, they can be viewed as "contingent" resources; an of-course operator on a resource in a context means that it will always be present in the context; that is, it is a "necessary" resource. The of-course operator attached to a goal ensures that the provability of the goal only depends on the necessary and not the contingent resources of the context. Thus, with respect to contingent resources, the goal $!(D \multimap G)$ behaves similarly to $demo(D, G)$.

We present two simple examples where this modality of !'ed formulas is illustrated. The first is a simple data base query program displayed in Figure 7. To make this example interesting, we have augmented the language with the `read`, `write`, and `nl` (new line) input/output commands. Figure 8 presents a session using this program. This example also shows some possible 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). Also, a command to retract a necessary (committed) entry can be executed without any problem (see Figure 8), but it does not have the effect of actually deleting the entry.

The kinds of manipulations demonstrated here can be used to correct the notion of state encapsulation and updating that was used in [15].

Our second example of the of-course modality is a simple natural language parsing example. In [23, 24] an intuitionistic context was used to manage the introduction and scoping of gaps. This approach, although modeling various aspects of gapping correctly, was unsatisfactory for at least two reasons. First, the phrase "whom Bob married Ann" would parse incorrectly as a relative clause. The restriction that a gap, once introduced, must be used is not easy to enforce using

```
db o- write('Command: ') x read(Command) x
      do(Command).
db o- write('Try again.') x nl x db.

do(enter(Entry))   o- entry(Entry) -o db.
do(commit(Entry))  o- bang(entry(Entry)) -o db.
do(retract(Entry)) o- entry(Entry) x db.
do(upd(Old,New)) o- entry(Old) x
                    (entry(New) -o db).
do(check(Q)) o- (entry(Q) x true x write(Q) x
            write(' is an entry.') x nl) & db.
do(necessary(Q)) o-
  (bang(entry(Q)) x true x write(Q) x
   write(' is a necessary entry') x nl) & db.
do(quit) o- true.
```

Figure 7: A Simple Data Base Query Program

```
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.
```

Figure 8: A Session Using the Data Base Query Program

```
sent(P1,P2) o- bang(np(P1,P0)) x vp(P0,P2).
vp(P1,P2)   o- tv(P1,P0) x np(P0,P2).
vp(P1,P2)   o- stv(P1,P0) x sbar(P0,P2).
sbar([that|P1],P2) o- sent(P1,P2).
np(P1,P2)   o- pn(P1,P2).
rel([whom|X],Y) o- all z\(np(z,z)) -o sent(X,Y).
pn([mary|L],L) o- 1.
pn([bob|L],L) o- 1.
pn([ann|L],L) o- 1.
tv([loves|L],L) o- 1.
tv([married|L],L) o- 1.
stv([believes|L],L) o- 1.
```

Figure 9: A simple parser for gaps in English

an intuitionistic context. Second, various restrictions on occurrences of gaps are not explained using such 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"). This "modal" distinction between these two kinds of noun phrases is not addressed naturally using intuitionistic logic.

The small logic program in Figure 9 illustrates how linear contexts can be used to solve these problems. Here, a *definite clause grammar (DCG)* [28] style presentation of a parser is used. Each category of the grammar, such as `sent` for sentence, `vp` for verb phrase, `sbar` for complement clauses, etc., is given two additional arguments, denoting a difference list of words. The rule for relative clauses (`rel`) introduces a gap, namely the formula `all z\(np(z,z))`. This formula represents a contingent resource: a gap of zero length. It can be used to prove the noun phrase mentioned in the `vp` definite formula but not the one in the `sent` definite formula: the latter occurrence of `np` is protected by a !. Thus the two goals

```
rel([whom,ann,believes,that,bob,married],[])
        rel([whom, bob, married],[])
```

are provable but the two goals

```
rel([whom,ann,believes,that,married,bob],[])
      rel([whom, bob, married, ann],[])
```

are not. As this parser rules out subject extraction, sentences that require such extractions must be handled with additional specialized grammer rules. Several similar types of "island constraints" occur in natural language parsing problems [27]. The use of !'ed formulas may aid in handling these constraints as well.

6 Related Work

There are many ways in which linear logic can be fruitfully exploited to address aspects of logic programming. Girard modeled 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 using a non-commutative linear logic [11]. Cerrito appplied classical linear logic to the problem of formalizing finite failure for certain kinds of Horn clause programs where negations are permitted in the body of clauses [5].

Linear logic has been used to extend the basic design of logic programming languages in at least two papers other than this one. Andreoli and Pareschi extended Horn clauses so that programs in the resulting language make use of the multiple conclusion nature of full linear logic [2]. In that extension, a form of context on the right of a sequent arrow is possible. They present several examples and argue that various aspects of object-oriented programming can be supported naturally within such contexts. Interestingly, their extension is rather different than ours: the intersection of the classes of program formulas in the two systems is just the set of Horn clauses. Harland and Pym also proposed In a fragment of linear logic as a logic programming language [13]. As was done here, the fragment is chosen so that uniform proofs remain complete. Since having !'s in succedents stops several inference rule permutations from holding, their proposal disallows such succedents. Thus, goal formulas are weaker than those presented here, but contexts are richer. The loss of !'ed goals, however, means that the examples in Section 5 cannot be coded directly.

In the area of natural language parsing, Lambek [16, 17] used a logic that can be identified with a non-commutative variant of linear logic for inferring the syntactic categories of phrases. Recently, Pereira handled gaps using a (commutative) linear logic-like context mechanism [26]. Neither of these approaches use & or the "of course" operator and, hence, the apparent modal distinction between noun phrase as subject or object cannot be captured directly in them.

7 Conclusion

There have been several examples in print of the need to refine the notion of intuitionistic context within logic programming. In this abstract, we proposed a

refinement using a fragment of linear logic. We argued that this fragment is a sensible logic programming language and presented an interpreter for it. Finally, we outlined how those problems with intuitionistic contexts can be addressed directly using the refinements available from linear logic.

A prototype interpreter, written in SML, of the first-order logic programming language described here is available from the first author.

Acknowledgements

The authors are grateful to Jean-Marc Andreoli, Gianluigi Bellin, Jawahar Chirimar, Remo Pareschi, and Fernando Pereira for helpful conversations regarding this work.

References

[1] Samson Abramsky. Computational interpretations of linear logic. Technical Report Research Report DOC 90/20, Imperial College, October 1990.

[2] J.-M. Andreoli and R. Pareschi. Linear objects: Logical processes with built-in inheritance. In *Proceeding of the Seventh International Conference on Logic Programming, Jerusalem*, May 1990.

[3] Anthony J. Bonner, L. Thorne McCarty, and Kumar Vadaparty. Expressing database queries with intuitionistic logic. In *Logic Programming: Proceeding of the North American Conference*, pages 831–850, 1989.

[4] Kenneth A. Bowen and Robert A. Kowalski. Amalgamating language and metalanguage in logic programming. In K.L. Clark and S.-A. Tarnlund, editors, *Logic programming*, volume 16 of *APIC studies in data processing*, pages 153 – 172. Academic Press, 1982.

[5] Serenella Cerrito. A linear semantics for allowed logic programs. In John Mitchell, editor, *Proceedings of the Fifth Annual Symposium on Logic in Computer Science, Philadelphia, PA*, pages 219 – 227, June 1990.

[6] Roy Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. Unpublished report from St. Andrews, Scotland, September 1990.

[7] Amy Felty. *Specifying and Implementing Theorem Provers in a Higher-Order Logic Programming Language*. PhD thesis, University of Pennsylvania, August 1989.

[8] Amy Felty and Dale Miller. Specifying theorem provers in a higher-order logic programming language. In *Ninth International Conference on Automated Deduction*, pages 61 – 80, Argonne, IL, May 1988. Springer-Verlag.

[9] D. M. Gabbay and U. Reyle. N-Prolog: An extension of Prolog with hypothetical implications. I. *Journal of Logic Programming*, 1:319 – 355, 1984.

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

[11] Jean-Yves Girard. Towards a geometry of interaction. In *Categories in Computer Science*, volume 92 of *Contemporary Mathematics*, pages 69 – 108. AMS, June 1987.

[12] Lars Hallnäs and Peter Schroeder-Heister. A proof-theoretic approach to logic programming. 1. Clauses as rules. *Journal of Logic and Computation*, pages 261–283, December 1990.

[13] James Harland and David Pym. The uniform proof-theoretic foundation of linear logic programming. Technical Report ECS-LFCS-90-124, Laboratory for the Foundations of Computer Science, University of Edinburgh, November 1990.

[14] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Second Annual Symposium on Logic in Computer Science*, pages 194–204, Ithaca, NY, June 1987.

[15] Joshua Hodas and Dale Miller. Representing objects in a logic programming language with scoping constructs. In David D. H. Warren and Peter Szeredi, editors, *1990 International Conference in Logic Programming*, pages 511–526. MIT Press, June 1990.

[16] J. Lambek. The mathematics of sentence structure. *American Mathematical Monthly*, 65:154 – 169, 1958.

[17] J. Lambek. Multicategories revisited. In *Categories in Computer Science*, volume 92 of *Contemporary Mathematics*, pages 217 – 239. AMS, June 1987.

[18] L. T. McCarty. Clausal intuitionistic logic I. fixed point semantics. *Journal of Logic Programming*, 5:1 – 31, 1988.

[19] Dale Miller. A logical analysis of modules in logic programming. *Journal of Logic Programming*, 6:79 – 108, 1989.

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

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

[22] Gopalan Nadathur and Dale Miller. An Overview of λProlog. In *Fifth International Logic Programming Conference*, pages 810–827, Seattle, Washington, August 1988. MIT Press.

[23] Remo Pareschi. *Type-driven Natural Language Analysis*. PhD thesis, University of Edinburgh, 1989.

[24] Remo Pareschi and Dale Miller. Extending definite clause grammars with scoping constructs. In David D. H. Warren and Peter Szeredi, editors, *1990 International Conference in Logic Programming*, pages 373–389. MIT Press, June 1990.

[25] Lawrence C. Paulson. Isabelle: The next 700 theorem provers. In Peirgiorgio Odifreddi, editor, *Logic and Computer Science*, pages 361 – 386. Academic Press, 1990.

[26] Fernando C. N. Pereira. Semantic interpretation as higher-order deduction. In *Proceedings of the Second European Workshop on Logics and AI*. Springer-Verlag, September 1990.

[27] Fernando C. N. Pereira and Stuart M. Shieber. *Prolog and Natural-Language Analysis*, volume 10. CLSI, Stanford, CA, 1987.

[28] Fernando C. N. Pereira and David H. D. Warren. Definite clauses for language analysis. *Artificial Intelligence*, 13:231 – 278, 1980.

[29] Frank Pfenning. Partial polymorphic type inference and higher-order unification. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 153–163, ACM Press, July 1988.

[30] Harold Schellinx. Some syntactical observations on linear logic. Technical Report ML-90-08, ITLI Prepublication Series for Mathematical Logic and Foundations, University of Amsterdam, 1990.