Monadic Concurrent Linear Logic Programming

Pablo López
Universidad de Málaga
lopez@lcc.uma.es

Frank Pfenning
Carnegie Mellon University
fp@cs.cmu.edu

Jeff Polakow
AIST, CVS, JST *
j-polakow@aist.go.jp

Kevin Watkins
Carnegie Mellon University
kw@cs.cmu.edu

ABSTRACT

Lolli is a logic programming language based on the asynchronous propositions of intuitionistic linear logic. It uses a backward chaining, backtracking operational semantics. In this paper we extend Lolli with the remaining connectives of intuitionistic linear logic restricted to occur inside a monad, an idea taken from the concurrent logical framework (CLF). The resulting language, called LolliMon, has a natural forward chaining, committed choice operational semantics inside the monad, while retaining Lolli's semantics outside the monad. LolliMon thereby cleanly integrates both concurrency and saturation with logic programming search. We illustrate its expressive power through several examples including an implementation of the pi-calculus, a call-by-need lambda-calculus, and several saturating algorithms presented in logical form.

1. INTRODUCTION

Combining computational paradigms, such as functional, logic, imperative, concurrent, constraint, or object-oriented programming, in a clean, uniform, and effective way is generally quite difficult because of the deep philosophical, theoretical, and pragmatic differences that divide them. Nevertheless, there have been many attempts to do so. One reason is that many algorithms and computational patterns that can be expressed naturally in one paradigm may be quite cumbersome in others. By combining paradigms we can hope to get the best of both worlds. An additional reason is that applications demand increasingly richer programming concepts. In particular, concurrent and distributed programming are becoming more prevalent, and therefore corresponding facilities are available in modern languages, many of which were not originally conceived with concurrency in mind. This is reflected in the difficulties we have in reasoning about programs written in these languages, especially when they employ concurrency using low-level primitives.

In this paper we advance the thesis that *logic* can be a powerful unifying force in the design of multi-paradigm languages. More concretely, we extend backward-chaining logic programming with *concurrency* and *saturation*, employing logical concepts such as lax truth and linearity in novel ways to obtain an elegant, minimalistic, yet general account of these computational phenomena.

One starting point for our design is the linear logic programming language Lolli [12]. It is based on a fragment of intuitionistic linear logic [9] endowed with an operational semantics in the form of backward-chaining search. It forms an *abstract logic programming language* [20] and therefore permits a natural interpretation of the logical connectives as goal-directed search instructions. In Andreoli's terminology [1] we say that Lolli is based on the *asynchronous connectives* of linear logic. Lolli programs use linear assumptions to encode state and thereby capture some aspects of imperative computation in a logical manner.

The second starting point is the concurrent logical framework CLF [4, 29, 30]. The aspect of CLF most relevant to this paper is that it extends the asynchronous fragment of intuitionistic linear logic with synchronous connectives, but restricts them to occur inside a *monad* [22]. From a logical perspective, this monad is a modal operator satisfying the laws of lax logic [5] in its judgmental formulation [24]. The present paper introduces a computational perspective: the monad represents a dividing line between the backward-chaining, backtracking semantics with asynchronous connectives outside the monad, and forward-chaining, committed choice semantics with synchronous connectives inside the monad.

The monadic encapsulation prevents undesirable interference between the two forms of computation and permits their harmonious co-existence. As we

*National Institute of Advanced Industrial Science and Technology (AIST), Research Center for Verification and Semantics (CVS), CREST, Japan Science and Technology Agency (JST)

This research has been supported by the Office of Naval Research (ONR) under grant MURI N00014-04-1-0724: Distributed System Security via Logical Frameworks.

will see, the two forms of computation are closely coupled and mutually dependent, yet predictable and clearly identifiable without destroying the useful properties of their components. It seems difficult if not impossible to achieve this without the use of a monad. Monads have been employed for similar reasons in the context of functional programming [28]. In logic programming, their only use we are aware of is for higher-order programming with data structures [3, 16], which is quite different from our application.

There have been many other studies of concurrent logic programming (see, for example, an early survey [27]), and concurrent logic programming in fragments of linear logic (for example, LO [2] and ACL [13]), but we are not aware of any that combine several forms of search in the same orthogonal way. Perhaps most closely related is Forum [18], which is based on classical linear logic, but to the authors' knowledge its operational semantics and the interaction between concurrent and sequential computation in Forum has never been fully clarified. Another notable attempt combining backward and forward chaining by Harland et al. [11] is an interesting foundational study of proof search, but it is not clear whether it can be realized in a practical logic programming language.

Our proposal has been implemented as a concrete language we call *LolliMon*. We have programmed numerous running examples, several of which are given in this paper. From these examples one can extract certain basic programming techniques, although we do not yet claim to have full understanding of the programming methodology or implementation challenges posed by our language.

One aspect that seems to be emerging is a close relationship between concurrent and saturating computation. The latter has been popular for some time in the treatment of constraints (see, for example, CHR [6]) and recently in the logical specification and complexity analysis of several algorithms [7, 15]. Most recently, the deletion of facts has been introduced into the logical algorithm framework. This appears to be modeled well by the consumption of linear assumptions in LolliMon.

The remainder of the paper is structured as follows. Section 2 introduces the syntax, judgments, and rules of the logic on which LolliMon is based. Section 3 describes the operational semantics of LolliMon. Section 4 presents several examples of LolliMon logic programs. Section 5 summarizes and sketches future work.

2. LOGICAL RULES

In this section we describe the basic syntactic structure of LolliMon, the logical rules on which its proof search strategy is based, and the basic aspects of the operational semantics that controls the application of the rules during proof search. Section 3 treats the design of the operational semantics and the interesting phenomena that arise in modeling concurrency and saturation more completely.

2.1 Syntax

The formula language of LolliMon consists of *asynchronous* connectives A, which it shares with the earlier logic programming language Lolli, and *synchronous* connectives S, which were not a part of Lolli's formula language. The distinction between synchronous and asynchronous connectives is fundamental; as we will see, the proof rules for asynchronous connectives have a natural logic programming interpretation in terms of backwards, goal-directed search (in the style pioneered by the original Prolog and formalized by the notion of *uniform proofs* [20]), while the synchronous connectives have a logic programming interpretation based on undirected, forward-chaining, concurrent execution.

The connection between these two rather different modes of execution is mediated by the monad constructor $\{\cdot\}$. Syntactically, the monad constructor allows synchronous connectives to be embedded within asynchronous formulas. Semantically, the point at which the monad constructor is introduced in a proof demarcates a transition from backward to forward reasoning (thinking, as usual, of the construction of the proof from the outermost goal upward).

The formula language of LolliMon is as follows.

$$\begin{array}{lcl} A & ::= & P \mid \top \mid A_1 \mathbin{\&} A_2 \mid \\ & & A_1 \multimap A_2 \mid A_1 \supset A_2 \mid \forall x{:}\tau.A \mid \{S\} \\ S & ::= & A \mid {!}\,A \mid \mathbf{1} \mid S_1 \otimes S_2 \mid \exists x{:}\tau.S \end{array}$$

As in other logic programming languages such as λProlog [19], object types τ are used to enforce the well-formedness of the objects of the system being modeled. The syntax of objects and object types is not discussed in the present paper; it is an entirely standard (prenex) polymorphically typed λ-calculus. The nonterminal P stands for an atomic proposition, which must be well-formed according to the discipline of the object type system.

In the presentation of our logic, we will make use of three kinds of formula context, Γ, Δ, and Ψ, which respectively contain unrestricted (intuitionistic) asynchronous hypotheses, linear asynchronous hypotheses, and linear synchronous hypotheses.

$$\begin{array}{lcll} \Gamma & ::= & \cdot \mid \Gamma, A & \text{Unrestricted context} \\ \Delta & ::= & \cdot \mid \Delta, A & \text{Linear context} \\ \Psi & ::= & \cdot \mid S, \Psi & \text{Synchronous context} \end{array}$$

For the purposes of this description we may take Γ and Δ to be multisets. We write Δ_1, Δ_2 to denote the multiset union of Δ_1 and Δ_2. There will be no explicit contraction or weakening rules for Γ, but contraction and weakening for Γ are metatheorems of LolliMon's sequent calculus.

In the LolliMon interpreter, the order of hypotheses is significant only in terms of which hypothesis is nondeterministically selected for focusing. However the third sort of context, Ψ, is not only linear but ordered; the ordering of Ψ helps to control the sequence of invertible left rules applied to its elements, as we will see below.

We present the logical rules of LolliMon in a sequent calculus formulation intended to bring out most clearly the relationship of the rules to LolliMon's operational semantics. For a natural-deduction-style presentation, see the technical report on CLF (the dependent type theory of which LolliMon's logic is a fragment) [29].

We classify the logical rules of LolliMon into three basic kinds: *inversion*, *focusing*, and *transition*. Inversion rules are always applied eagerly; their use can never cause proof search to fail. Focusing rules are those which entail a significant choice; their use can cause a particular branch in proof search to fail. All inversion and focusing rules act either on a specific hypothesis (left) or on the conclusion (right) of the sequent, and we identify them accordingly. Lastly, we have transition rules that initiate or terminate a sequence of inversion or focusing rules.

Very roughly, the operational semantics of LolliMon can be described in terms of five modes of execution. Four capture inversion and focusing, each on the left and the right. An additional mode is needed because of differences in the way left focusing happens depending on whether the formula on the right is in the monad (a synchronous formula S) or outside it (an atomic formula P).

Our first two computation modes, *right inversion* and *left focusing*, are associated with asynchronous formulas A, and are inherited from the operational semantics of Lolli [12].

$\Gamma; \Delta \Rightarrow A$	Right inversion
$\Gamma; \Delta; A \gg P$	Left focusing

The other three computation modes, *monadic left focusing*, *left inversion*, and *right focusing*, are associated with synchronous formulas S. In addition, we have a transition sequent, which, when read bottom-up, marks the end of left inversion and precedes monadic left or right focusing. We call this a *forward chaining* sequent in view of its eventual operational interpretation. The transition on the other side, from right inversion to left focusing, takes place at $\Gamma; \Delta \Rightarrow P$. There is no separate sequent form for the latter transition, because there only one kind of focusing is possible.

$\Gamma; \Delta \rightarrow S$	Forward chaining
$\Gamma; \Delta; A > S$	Monadic left focusing
$\Gamma; \Delta; \Psi \rightarrow S$	Left inversion
$\Gamma; \Delta \gg S$	Right focusing

The operational meaning of these modes is discussed below, along with the logical rules for each of them.

2.2 Asynchronous Formulas

As LolliMon is a conservative extension of Lolli, which is based on the asynchronous connectives $\top$, $\&$, $\supset$, $\multimap$, and $\forall$, the logical rules for these connectives are inherited essentially unchanged from Lolli. For each connective, there is a right inversion rule and a left focusing rule. The basic search strategy decomposes the right formula using inversion rules until it is atomic, then non-deterministically selects a hypothesis, which is focused on until the same atomic formula is reached. The following rules are used to select an unrestricted or a linear hypothesis for focusing.

$$\frac{\Gamma, A; \Delta; A \gg P}{\Gamma, A; \Delta \Rightarrow P}\ \text{uhyp} \qquad \frac{\Gamma; \Delta; A \gg P}{\Gamma; \Delta, A \Rightarrow P}\ \text{lhyp}$$

The following axiom rule says we are finished if the atomic formula from the goal matches the atomic formula obtained by focusing on a hypothesis.

$$\frac{}{\Gamma; \cdot; P \gg P}\ \text{atm}$$

The right and left rules for unrestricted implication are standard.

$$\frac{\Gamma, A; \Delta \Rightarrow B}{\Gamma; \Delta \Rightarrow A \supset B}\ {\supset_R} \qquad \frac{\Gamma; \Delta; B \gg P \qquad \Gamma; \cdot \Rightarrow A}{\Gamma; \Delta; A \supset B \gg P}\ {\supset_L}$$

Note the restriction in $\supset_L$ that the derivation of A not require linear hypotheses.

The right and left rules for linear implication are also standard:

$$\frac{\Gamma; \Delta, A \Rightarrow B}{\Gamma; \Delta \Rightarrow A \multimap B}\ {\multimap_R} \qquad \frac{\Gamma; \Delta_1; B \gg P \quad \Gamma; \Delta_2 \Rightarrow A}{\Gamma; \Delta_1, \Delta_2; A \multimap B \gg P}\ {\multimap_L}$$

The right and left rules for additive conjunction and unit are as follows:

$$\frac{}{\Gamma; \Delta \Rightarrow \top}\ \top_R \qquad \text{(no } \top_L \text{ rule)}$$

$$\frac{\Gamma; \Delta \Rightarrow A \qquad \Gamma; \Delta \Rightarrow B}{\Gamma; \Delta \Rightarrow A \,\&\, B}\ \&_R$$

$$\frac{\Gamma; \Delta; A \gg P}{\Gamma; \Delta; A \,\&\, B \gg P}\ \&_{L1} \qquad \frac{\Gamma; \Delta; B \gg P}{\Gamma; \Delta; A \,\&\, B \gg P}\ \&_{L2}$$

Finally, we have right and left rules for universal quantification.

$$\frac{\Gamma; \Delta \Rightarrow [a/x]A}{\Gamma; \Delta \Rightarrow \forall x{:}\tau.A}\ \forall_R \qquad \frac{\Gamma; \Delta; [t/x]A \gg P}{\Gamma; \Delta; \forall x{:}\tau.A \gg P}\ \forall_L$$

Here, a stands for a fresh parameter of type τ, the scope of which is restricted to the subderivation above the $\forall_R$ rule. The term t in the corresponding left rule must be well-formed of type τ with respect to all parameters that are in scope at the point the left rule is applied. At the level of description considered here, the term t is picked non-deterministically. In the LolliMon implementation, it is determined by unification.

This concludes the description of the fragment of LolliMon inherited from Lolli.

There is one additional rule associated with right inversion: the right rule for the monad constructor $\{\cdot\}$.

$$\frac{\Gamma; \Delta \rightarrow S}{\Gamma; \Delta \Rightarrow \{S\}}\ \{\}_R$$

This rule moves the proof search procedure from the goal-directed inversion mode of Lolli to the forward-chaining mode of LolliMon. The logical interpretation of the forward chaining judgment appearing in the premise is the lax judgment of lax logic [5, 24]. Because the monad constructor connects the truth judgment and the lax judgment of the logic, it is subject to somewhat different symmetries than the rest of the asynchronous connectives. In particular, its left rule does not appear among the left focusing rules treated in this section; instead, it appears in the monadic left focusing rules discussed in Section 2.3.

2.3 Synchronous Formulas

We now turn to the proof rules concerning the synchronous connectives $\otimes$, $\mathbf{1}$, $\exists$, and !.

Here, the right rules are not invertible, so the strategy of first decomposing the formula on the right by inversion is inadequate. In general, the derivation of a sequent with a synchronous goal formula S must first allow for some forward reasoning from hypotheses before applying non-invertible right rules to break down the goal S; see Andreoli's analysis of linear logic proof search [1] for more details. Exactly which hypotheses are focused on for forward reasoning is a non-deterministic choice.

We have the following two rules, which allow for focusing on an unrestricted or linear hypothesis.

$$\frac{\Gamma, A; \Delta; A > S}{\Gamma, A; \Delta \rightarrow S}\ \mathrm{uhyp}' \qquad \frac{\Gamma; \Delta; A > S}{\Gamma; \Delta, A \rightarrow S}\ \mathrm{lhyp}'$$

Here the special nature of the lax judgment underlying the forward chaining mode enters: the hypothesis selected must have a head of the form $\{S'\}$, instead of a head of the form P. This requirement is made explicit in the $\{\}_L$ rule below. There is additional nondeterminism here, compared with the case of heads of the form P, because the formula S' need not match the ultimate goal formula S on the right.

The monad constructor $\{S'\}$ is decomposed by the following left rule.

$$\frac{\Gamma; \Delta; S' \rightarrow S}{\Gamma; \Delta; \{S'\} > S}\ \{\}_L$$

After removing the monad constructor, the formula S' is decomposed by left rules, which *are* invertible for synchronous formulas. These left rules can add further parameters and hypotheses to the contexts. Finally, once S' is completely decomposed, we return to the forward chaining judgment:

$$\frac{\Gamma; \Delta \rightarrow S}{\Gamma; \Delta; \cdot \rightarrow S}\ \rightarrow\rightarrow$$

Thus, the overall effect of the sequence of rules beginning with uhyp$'$ or lhyp$'$ and ending with $\rightarrow\rightarrow$ is to consume some hypotheses during left focusing by propagating them to subgoals, and introduce others by left inversion. The goal formula S on the right is left alone. We think of this sequence as an atomic step in the execution of a concurrent system. The application of the $\rightarrow\rightarrow$ rule demarcates the atomic step. As described in Section 3, the use of the $\rightarrow\rightarrow$ rule forces a commitment to all of the suspended non-deterministic choices encountered during the atomic step.

We have seen how the forward chaining judgment allows working on the left by reasoning forward from hypotheses. The other way to proceed, when forward chaining, is to choose to work on the goal S on the right, by applying non-invertible right rules. Operationally, the decision to focus on the goal on the right implicitly terminates forward chaining mode. We capture this behavior with the following rule.

$$\frac{\Gamma; \Delta \gg S}{\Gamma; \Delta \rightarrow S}\ \gg\rightarrow$$

This rule corresponds to the coercion from the truth judgment to the lax judgment in lax logic.

Note that all three forward chaining rules, uhyp$'$, lhyp$'$, and $\rightarrow\rightarrow$, are applicable whenever the search procedure is in forward chaining mode. For the purposes of the logical description of LolliMon, we are content to leave as a non-deterministic choice which forward chaining rule should be applied. The operational semantics described in Section 3 imposes further structure on these choices.

2.3.1 *Monadic Left Focusing*

The rules associated with the monadic left focusing judgment $\Gamma; \Delta; A > S$ are very similar to those described previously for $\Gamma; \Delta; A \gg P$. In both cases, non-invertible left rules are applied to the formula A until its head ($\{S'\}$ or P', respectively) is reached. However, the two sequent forms differ, in that the head P' of the latter is forced to be equal to the atomic goal P produced by right inversion, while the head $\{S'\}$ of the former does not have to be equal to the formula S on the right.

Here we show the modified left rule for linear implication; the other left rules for asynchronous connectives are modified along the same lines.

$$\frac{\Gamma; \Delta_2; B > S \qquad \Gamma; \Delta_1 \Rightarrow A}{\Gamma; \Delta_1, \Delta_2; A \multimap B > S}\ \multimap_L'$$

2.3.2 *Left Inversion*

The rules for invertibly decomposing a synchronous formula on the left are simple adaptations of the usual sequent calculus left rules for each connective to our setting. When inversion begins, the context Ψ contains a single formula S'. The context Ψ can contain more than one formula after some left rules have been applied. Invertibility allows us to force the decomposition to occur in a left-to-right order. This avoids permutations of the left rules that would otherwise lead to an unwanted explosion in the number of proofs. Thus, each rule decomposes the leftmost synchronous formula in the context Ψ. When an asynchronous formula becomes leftmost, it is moved to the linear context, because there are no more invertible left rules to apply to it. Finally, at some point the context Ψ becomes empty, and forward chaining resumes.

The left rules for multiplicative conjunction and unit are as follows:

$$\frac{\Gamma; \Delta; \Psi \rightarrow S}{\Gamma; \Delta; \mathbf{1}, \Psi \rightarrow S}\ \mathbf{1}_L \qquad \frac{\Gamma; \Delta; S_1, S_2, \Psi \rightarrow S}{\Gamma; \Delta; S_1 \otimes S_2, \Psi \rightarrow S}\ \otimes_L$$

The left rule for the existential quantifier is the following:

$$\frac{\Gamma; \Delta; [a/x]S', \Psi \rightarrow S}{\Gamma; \Delta; \exists x{:}\tau.S', \Psi \rightarrow S}\ \exists_L$$

As in the $\forall_R$ rule, the parameter a has type τ and its scope is the subderivation above the $\exists_L$ rule.

The left rule for the unrestricted modality adds a hypothesis to the unrestricted context. The syntax of LolliMon requires the formula underneath the modality to be asynchronous.

$$\frac{\Gamma, A; \Delta; \Psi \rightarrow S}{\Gamma; \Delta; !\, A, \Psi \rightarrow S}\ !_L$$

Finally, the rule for moving an asynchronous formula to the linear context is as follows:

$$\frac{\Gamma; \Delta, A; \Psi \rightarrow S}{\Gamma; \Delta; A, \Psi \rightarrow S}\ \text{async}$$

2.3.3 Right Focusing

At some point the derivation transitions from forward chaining to right focusing mode, which applies non-invertible right rules to search for a proof of the synchronous goal formula. Each of these rules is a simple adaptation of the corresponding right rule from linear logic.

The right rules for multiplicative conjunction and unit are as follows:

$$\frac{}{\Gamma; \cdot \gg \mathbf{1}}\ \mathbf{1}_R \qquad \frac{\Gamma; \Delta_1 \gg S_1 \qquad \Gamma; \Delta_2 \gg S_2}{\Gamma; \Delta_1, \Delta_2 \gg S_1 \otimes S_2}\ \otimes_R$$

The right rule for the existential quantifier is the following:

$$\frac{\Gamma; \Delta \gg [t/x]S}{\Gamma; \Delta \gg \exists x{:}\tau.S}\ \exists_R$$

As in the $\forall_L$ rule, we require t to have type τ and determine it by unification.

The right rule for the unrestricted modality is the following:

$$\frac{\Gamma; \cdot \Rightarrow A}{\Gamma; \cdot \gg !\, A}\ !_R$$

The syntax of LolliMon requires an asynchronous formula under the unrestricted modality, so at the point the right rule for the modality is applied, the right focusing stage ends.

Finally, when an asynchronous formula appears as a subformula of a synchronous formula, the right focusing stage ends, and a new right inversion stage begins.

$$\frac{\Gamma; \Delta \Rightarrow A}{\Gamma; \Delta \gg A}\ \Rightarrow\gg$$

The logical rules of LolliMon enjoy the admissibility of cut. This is a consequence of a corresponding theorem for the natural deduction formulation of CLF [29].

3. OPERATIONAL SEMANTICS

Designing an operational semantics for the proof rules given in Section 2 is non-trivial. There is a great deal of non-determinism to be removed, and there are many choices to be taken, potentially leading to different operational behavior. In this section, we informally describe an operational semantics for LolliMon which results in a language integrating both committed choice concurrency and backtracking goal-directed search.

A number of design decisions guided our specification. Since LolliMon is conceived as a concurrent logic programming language, completeness of the proof search procedure is not a concern. More concretely, committed choice and quiescence (to be described below) lead to a strategy that will not explore all of the search space. In addition, the operational semantics must be compatible with Lolli's over the fragment that Lolli and LolliMon share.

The operational semantics of LolliMon is a bidirectional proof search procedure. When LolliMon proofs contain no monadic goals, the proof search procedure is similar to that of other logic languages such as λProlog or Lolli: a backward-chaining, backtracking search for a uniform proof. In contrast, when a monadic goal is found, a forward-chaining, committed choice strategy is applied. Thus the monad affects the basic mode of proof search, and can be interpreted as staging the LolliMon computation. In the remainder of this section we detail the different stages of this computation: goal-directed search, backward-chaining and forward-chaining.

3.1 Example: Asynchronous π-Calculus

In order to make the discussion of LolliMon's operational semantics more concrete, we will be using a representation of an asynchronous[1] variant of Milner's π-calculus [21] as a running example. The π-calculus is based on a syntax for concurrently executing *processes* P. The processes interact by sending and receiving messages along named *channels* c.

The basic process constructors are parallel composition $P \mid Q$ and its unit 0 (representing a process that is finished). There is also a construct $\nu c.P(c)$ for binding a new channel name within a process. In traditional presentations of the π-calculus, scope extrusion is used to propagate the bound channel name from its initial scope to more global scopes. In LolliMon, we will see that fresh parameters introduced by the $\exists_L$ rule fulfill the same function.

The process constructors concerned with communication are $c(x).P(x)$ for inputting a value along the channel c (with the process $P(x)$ continuing to execute using the communicated value in place of the variable x), and $\bar{c}\langle v\rangle$ for outputting the value v along the channel c. In this simple variant of the π-calculus, the only values are the channel names themselves.

In addition, there is a replicated variant $!c(x).P(x)$ of the input construction, which allows arbitrarily many values to be received along channel c, each spawning a new copy of $P(x)$ with the received value subsituted for x. The replicated input process itself persists forever, once it has been introduced.

This concludes the syntax of the π-calculus. In the LolliMon implementation, each of these process constructors is represented by a term constructor in LolliMon's term language.

expr : **type**. **chan** : **type**.

par : **expr** → **expr** → **expr**.
zero : **expr**.
new : (**chan** → **expr**) → **expr**.

[1]The use of "asynchronous" here is not related to our earlier uses of the term.

$\mathbf{in} : \mathbf{chan} \to (\mathbf{chan} \to \mathbf{expr}) \to \mathbf{expr}.$
$\mathbf{rin} : \mathbf{chan} \to (\mathbf{chan} \to \mathbf{expr}) \to \mathbf{expr}.$
$\mathbf{out} : \mathbf{chan} \to \mathbf{chan} \to \mathbf{expr}.$

Note the use of higher-order abstract syntax to represent the π-calculus binding constructs. There is a bijective *representation function* $\ulcorner \cdot \urcorner$ mapping π-calculus process expressions to well-typed, canonical LolliMon terms having type **expr**.

$$\begin{aligned}
&\ulcorner P \mid Q \urcorner = \mathbf{par} \ulcorner P \urcorner \ulcorner Q \urcorner \\
&\ulcorner 0 \urcorner = \mathbf{zero} \\
&\ulcorner \nu c.P(c) \urcorner = \mathbf{new}\,(\lambda c{:}\mathbf{chan}.\ulcorner P(c) \urcorner) \\
&\ulcorner c(x).P(x) \urcorner = \mathbf{in}\,c\,(\lambda x{:}\mathbf{chan}.\ulcorner P(x) \urcorner) \\
&\ulcorner !c(x).P(x) \urcorner = \mathbf{rin}\,c\,(\lambda x{:}\mathbf{chan}.\ulcorner P(x) \urcorner) \\
&\ulcorner \bar{c}\langle v \rangle \urcorner = \mathbf{out}\,c\,v
\end{aligned}$$

3.2 Goal-Directed Proof Search

The right inversion rules of LolliMon are exactly those of Lolli, except for $\{\}_R$. These rules are applied from the bottom up to decompose a goal into atomic and/or monadic subgoals. It should be noted that rules other than the right rules are only considered when the goal is either atomic or monadic. When there are no monadic goals involved, LolliMon proofs are uniform [20]. When there are monadic goals, this strategy can be seen as an extension of uniform proofs, where the monad introduces an intermediate stage, forward chaining, before the goal is further decomposed. Forward chaining, which corresponds to the lax judgment of lax logic, is particular to LolliMon.

The non-determinism implicit in the choice of a term t in the $\exists_R$ rule (as well as the $\forall_L$ rule) is handled as usual by generating a logic variable and relying upon unification to appropriately instantiate this variable. Since our term calculus is a λ-calculus, we use a deterministic version of higher-order unification that suspends any non-pattern [17] unification problems until they are further instantiated. Any suspended unification problems remaining at the end of a query are reported as leftover constraints.

The non-deterministic splitting of the linear context implicit in the $\otimes_R$, $\multimap_L$, and $\multimap'_L$ rules is common to traditional linear logic programming languages [12, 25]. The low-level management of linear hypotheses turns out to be unaffected by the addition of a forward chaining phase to proof search. Our strategy for managing linear hypotheses is a new variation of the I/O system for Lolli [12]. The new strategy is described in detail in [14].

3.3 Backward Chaining

When an atomic goal is encountered in right inversion mode, a hypothesis with an atomic head must be chosen to focus on. Following the behavior of Lolli, our implementation keeps track of the order in which hypotheses[2], both linear and unrestricted, were assumed and tries to focus first on the most recent ones whose heads unify with the atomic goal; see Section 3.7 for more discussion on clause ordering in LolliMon. The usual Prolog backtracking semantics is used when a particular choice fails, or to search for alternative solutions.

Upon choosing a hypothesis, the system switches to left focusing mode and applies left rules to the selected hypothesis, keeping track of any new subgoals generated (by the $\supset_L$ and $\multimap_L$ rules) along the way. When the head of the selected hypothesis is finally exposed, it is unified with the original atomic goal, and unless unification fails, proof search will continue by solving any pending subgoals. This strategy is the usual backchaining mechanism of Prolog.

If left focusing encounters the $\&$ connective, two rules, $\&_{L1}$ and $\&_{L2}$, might be applied. The resolution of this choice is made by the same mechanism that chooses a hypothesis to focus on in the first place, as though there were two clauses, one for each branch of the $\&$.

3.4 Forward Chaining

As mentioned in Section 2.2, in order to construct a proof of $\{S\}$, the LolliMon system enters a forward chaining mode and tries to derive $\Gamma; \Delta \to S$. The most important difference between goal-directed search (as in Lolli) and forward chaining is that for forward chaining, the formula S on the right does not have to match the head of a clause selected for focusing on the left. In fact, it is perfectly acceptable for a program clause to be selected that doesn't mention S at all. However, only program clauses having *some* monadic formula $\{S'\}$ at their heads can be selected.

As Section 2.3 hints, we think of the forward chaining phase of proof search as a sequence of atomic steps. Each atomic step begins with an application of the uhyp$'$ or lhyp$'$ rule that selects a hypothesis A to focus on, leading to a sequent of the form

$$\Gamma; \Delta; A > S.$$

Next, the system breaks down A by applying a series of non-invertible left rules, postponing any new subgoals in the manner described in Section 3.3. Eventually, the head of A, which must be a formula of the form $\{S'\}$, is reached:

$$\Gamma; \Delta'; \{S'\} > S.$$

Here Δ' is the subset of Δ remaining after the context is split between this branch of the proof and any subgoals. The LolliMon implementation determines Δ' lazily.

At this point, after establishing that the chosen clause is indeed monadic, any pending subgoals are solved. If a subgoal cannot be solved, then the system backtracks to the beginning of the forward chaining cycle and looks for a new clause to focus on. The procedure up to this point is the same as that employed for backward chaining, except that no unification is involved.

Next, supposing any pending subgoals have been solved, the $\{\}_L$ rule is applied, leading to

$$\Gamma; \Delta'; S' \to S,$$

and after applying a series of invertible left rules to S', we have a sequent of the form

$$\Gamma, \Gamma_0; \Delta', \Delta_0; \cdot \to S$$

[2]We think of program clauses as hypotheses assumed at the beginning of each query.

where Γ_0 and Δ_0 are any new assumptions added during the decomposition of S'. Finally, the rule $\twoheadrightarrow$ is applied to finish the atomic step, and we are left with

$$\Gamma, \Gamma_0; \Delta', \Delta_0 \to S,$$

which is again in the forward chaining mode. The application of $\twoheadrightarrow$ additionally causes the system to commit to this atomic step by removing all the choice points generated while focusing on the original A or solving subgoals generated during focusing.

Thus, the overall effect of the atomic step is to possibly consume some linear hypotheses and possibly introduce new unrestricted or linear hypotheses. In addition, the process of proving the atomic step may have had a side effect on the state of the unification store.

The other way to proceed when in forward chaining mode

$$\Gamma; \Delta \to S$$

is to focus on the right-hand formula S using the $\gg\to$ rule, leading to

$$\Gamma; \Delta \gg S.$$

This terminates the forward chaining phase, and the LolliMon interpreter proceeds by applying non-invertible right rules to S. Sections 3.5 and 3.6 describe strategies for deciding when to end forward chaining in this way.

EXAMPLE. Using the syntax introduced earlier for the π-calculus, we can set up a forward-chaining interpreter for π-calculus processes. First, we introduce the following predicates:

$$\begin{array}{l} \mathbf{proc} : \mathbf{expr} \to \mathbf{o}. \\ \mathbf{msg} : \mathbf{chan} \to \mathbf{chan} \to \mathbf{o}. \end{array}$$

These two predicates correspond to the message and process types of Kobayashi and Yonezawa's ACL [13]. The idea of the interpreter is that the state of the π-calculus execution will be represented by the multiset of linear hypotheses available in forward chaining mode. The goal on the right-hand side while the interpreter is running will always be $\mathbf{1}$. Suppose we want to initialize the interpreter with a process

$$\begin{array}{rcl} P_0 & = & \ulcorner \bar{c}\langle x\rangle \mid c(y).0 \urcorner \\ & = & \mathbf{par}\,(\mathbf{out}\,c\,x)\,(\mathbf{in}\,c\,(\lambda y.\mathbf{zero})). \end{array}$$

Initially, we have only the linear hypothesis $\mathbf{proc}\,P_0$ in the context. By adding the rule

$$A_0 = \forall P.\forall Q.\mathbf{proc}\,(\mathbf{par}\,P\,Q) \multimap \{\mathbf{proc}\,P \otimes \mathbf{proc}\,Q\}$$

to our logic program, we enable a forward chaining step that consumes the hypothesis $\mathbf{proc}\,P_0$ and introduces two new hypotheses, namely $\mathbf{proc}\,(\mathbf{out}\,c\,x)$ and $\mathbf{proc}\,(\mathbf{in}\,c\,(\lambda y.\mathbf{zero}))$.

Suppose, then, that we start with just P_0 running, which according to the invariant of the interpreter, means that we are searching for a LolliMon proof of the sequent

$$\Gamma; \mathbf{proc}\,P_0 \to \mathbf{1}.$$

Here Γ includes all the rules of our logic program for the interpreter. The first step is to focus on the program clause A_0 above, using the uhyp$'$ rule, leaving

$$\Gamma; \mathbf{proc}\,P_0; A_0 > \mathbf{1}.$$

At this point we need to apply the rule $\forall'_L$ twice to get rid of the universal quantifiers at the outside of A_0. The LolliMon system will instantiate the bound variables P and Q of the universal quantifiers by unification, but we know what the instantiation will ultimately be, namely $P = \mathbf{out}\,c\,x$ and $Q = \mathbf{in}\,c\,(\lambda y.\mathbf{zero})$, so for this discussion we use it from the start.

We are then left with the sequent

$$\Gamma; \mathbf{proc}\,P_0; \mathbf{proc}\,(\mathbf{par}\,P\,Q) \multimap \{\mathbf{proc}\,P \otimes \mathbf{proc}\,Q\} > \mathbf{1}$$

with P and Q as above. The next step is to use $\multimap'_L$ to eliminate the linear implication. This leaves us with

$$\Gamma; \cdot; \{\mathbf{proc}\,P \otimes \mathbf{proc}\,Q\} > \mathbf{1}$$

while the second premise

$$\Gamma; \mathbf{proc}\,P_0 \Rightarrow \mathbf{proc}\,(\mathbf{par}\,P\,Q)$$

is stored on the subgoal stack. Here we've also anticipated the partitioning the linear hypotheses in the only way that can succeed: namely, the single linear hypothesis $\mathbf{proc}\,P_0$ is allocated to the second subgoal.

Since we now have formula of the form $\{S'\}$ on the left, we suspend left focusing and apply inversion to the stored subgoal:

$$\Gamma; \mathbf{proc}\,P_0 \Rightarrow \mathbf{proc}\,(\mathbf{par}\,P\,Q),$$

which is quickly dealt with, since $P_0 = \mathbf{par}\,P\,Q$.

We then return to the main focusing proof. The only rule that can be used is $\{\}_L$, which leads to

$$\Gamma; \cdot; \mathbf{proc}\,P \otimes \mathbf{proc}\,Q \to \mathbf{1}.$$

At this point, several left inversion rules are applied, and we end up with

$$\Gamma; \mathbf{proc}\,P, \mathbf{proc}\,Q; \cdot \to \mathbf{1}.$$

Now the only applicable rule is $\twoheadrightarrow$, which leaves us with

$$\Gamma; \mathbf{proc}\,P, \mathbf{proc}\,Q \to \mathbf{1}$$

to prove. There were no non-deterministic choices involved in the execution of this atomic step, because our program had only one clause, but in general, there might be choice points created between the use of uhyp$'$ or lhyp$'$ and the use of $\twoheadrightarrow$.

It is perhaps worth noting at this point that we are not at all interested in the question of *completeness*; that is, the possibility of finding any proof admitted by the logical rules. Such a notion of completeness might be useful in model checking, or some other analysis intended to explore the entire state space of the concurrent system. But LolliMon is not a model checker; the aim here is to *execute* a concurrent system, not to map its entire state space. Accordingly, just after each forward step is taken, any choice points introduced during the the proof of the forward step are dropped; LolliMon *commits* to the forward step. Also, once the decision to stop forward chaining has been taken, LolliMon commits to that decision, never going back to consider executions that might have lasted longer.

3.5 Quiescence

We have seen how LolliMon takes each atomic step in forward chaining mode. There remain the issues of *which* forward chaining steps it should take, and *when* to stop taking forward chaining steps and return to a goal-directed search strategy. The underlying logic on which LolliMon is based does not constrain these choices at all. In this section, and the following, we discuss the termination criterion, the decision about when to finish forward chaining and return to goal-directed proof search. Section 3.7 treats the question of which forward steps should be taken.

Since the intended semantics of forward chaining mode is the simulation of various concurrent object systems, the decision about when to finish forward chaining is equivalent to deciding how to terminate a concurrent system. The strategies which turn out to be most useful in practice are based purely on the behavior of the concurrent system; that is, they are based only on what forward steps are available to be taken. One might also imagine a criterion based on the goal formula S that will be proved on the right once forward chaining ends, but this turns out to be fraught with difficulties. The current LolliMon implementation never looks at the formula on the right when in forward chaining mode, either to decide which forward steps should be taken, or to decide when to stop taking forward steps. This reduces further the space of possible proofs found by the LolliMon interpreter.

The simplest strategy depending only on what forward steps are available to be taken is *quiescence*: the concurrent execution (that is, the forward chaining phase) ends when *no* forward step is available to be taken.

EXAMPLE. We can illustrate the notion of quiescence by completing the simple π-calculus execution begun above. First, we reveal the remaining clauses in the π-calculus interpreter's logic program:

$$\begin{array}{l}
\mathbf{proc\,zero} \multimap \{\mathbf{1}\}.\\
\forall P.\mathbf{proc}\,(\mathbf{new}\,(\lambda c.P\,c)) \multimap \{\exists c.\mathbf{proc}\,(P\,c)\}.\\
\forall C.\forall V.\mathbf{proc}\,(\mathbf{out}\,C\,V) \multimap \{\mathbf{msg}\,C\,V\}.\\
\forall C.\forall P.\mathbf{proc}\,(\mathbf{in}\,C\,(\lambda x.P\,x))\\
\qquad \multimap \{\forall V.\mathbf{msg}\,C\,V \multimap \{\mathbf{proc}\,(P\,V)\}\,\}.\\
\forall C.\forall P.\mathbf{proc}\,(\mathbf{rin}\,C\,(\lambda x.P\,x))\\
\qquad \multimap \{!\,\forall V.\mathbf{msg}\,C\,V \multimap \{\mathbf{proc}\,(P\,V)\}\,\}.
\end{array}$$

The first two clauses are similar to the clause we have already seen for processes of the form **par** $P\,Q$. In each case a process is interpreted by LolliMon's own logical connectives—concurrent composition becomes $\otimes$, the unit **zero** of composition becomes **1**, and the channel-binding construct becomes $\exists$. In the last case, the $\exists_L$ rule will introduce a fresh parameter to stand for the name of the new channel.

The final three clauses are concerned with asynchronous communication. Each message containing the value V and flowing along a channel C is represented by a linear hypothesis of the form $\mathbf{msg}\,C\,V$. The clause for **out** creates such messages. The clauses for **in** and **rin** use the powerful technique of *forward-generating new clauses*. In this case, the clause

$$\forall V.\mathbf{msg}\,C\,V \multimap \{\mathbf{proc}\,(P\,V)\}$$

is introduced as a new linear (for **in**) or unrestricted (for **rin**) hypothesis.

Continuing the example execution of Section 3.4, the sequent

$$\Gamma; \mathbf{proc}\,(\mathbf{out}\,c\,x), \mathbf{proc}\,(\mathbf{in}\,c\,(\lambda y.\mathbf{zero})) \rightarrow \mathbf{1},$$

after several atomic steps, will eventually reach

$$\Gamma; \mathbf{msg}\,c\,x, (\forall V.\mathbf{msg}\,c\,V \multimap \{\mathbf{proc\,zero}\}) \rightarrow \mathbf{1}.$$

At this point, none of the clauses in Γ can be successfully focused on, so the only choice is to use the second linear hypothesis. This leads to

$$\Gamma; \mathbf{msg}\,c\,x; \forall V.\mathbf{msg}\,c\,V \multimap \{\mathbf{proc\,zero}\} > \mathbf{1},$$

and the atomic step is finished by applying left rules and solving the subgoal $\mathbf{msg}\,c\,x$ using the other linear hypothesis. At this point we have

$$\Gamma; \mathbf{proc\,zero} \rightarrow \mathbf{1},$$

and finally, after one more atomic step, we have

$$\Gamma; \cdot \rightarrow \mathbf{1},$$

at which point no atomic forward chaining step is possible, so quiescence has been reached. Accordingly, the LolliMon system ends the forward chaining phase by applying $\gg\rightarrow$, and the proof is completed with 1_R.

3.6 Saturation

The last section introduced quiescence, defined as a state in which no further forward steps are possible. But sometimes it is more useful to consider *saturation*: a state in which forward steps may be possible, but they do not lead to any new information.

For example, we may want to execute *bottom-up logic programs* that repeatedly apply forward reasoning steps to a set of facts, each step extending the set of known facts, until no further facts can be deduced from those that are already known.

EXAMPLE. A simple example is a program to compute the transitive closure of a finite relation. Suppose we have a finite number of unrestricted hypotheses of the form $\mathbf{r}\,A\,B$ for various pairs A and B. We can then generate the transitive closure $\mathbf{rr}\,A\,B$ by the following program.

$$\begin{array}{l}
\forall A.\forall B.\mathbf{r}\,A\,B \supset \{!\mathbf{rr}\,A\,B\}.\\
\forall A.\forall B.\forall C.\mathbf{rr}\,A\,B \supset \mathbf{rr}\,B\,C \supset \{!\mathbf{rr}\,A\,C\}.
\end{array}$$

Each forward step taken by this program uses some facts concerning **r** and **rr** to derive a new fact about **rr** and add it to the unrestricted context. If we ran the program under the quiescence criterion for termination, it would execute forever (assuming we start with at least one fact concerning **r**), because more forward steps can always be taken, even if they only regenerate facts that were already known.

The saturation criterion, on the other hand, disallows forward reasoning steps that either have no effect on the set of available linear and unrestricted hypotheses, or simply reintroduce unrestricted hypotheses that were already known. We define a *non-trivial step* to be an atomic forward chaining step which causes a change to either of the logical contexts, or to the state of the unification store. We then define saturation to mean that no non-trivial step is possible.

Assuming there is at least one axiom concerning **r**, the program above saturates in a finite number of steps, because all the hypotheses introduced are unrestricted and **rr** is finite. Once every deducible consequence **rr** $A\,B$ ends up in the unrestricted context, no non-trivial steps are possible, and saturation is reached. We exhibit realistic examples of this bottom-up technique in Section 4.

The LolliMon implementation always uses saturation to decide when to stop forward chaining. This is not observably different from quiescence, except that a concurrent execution might run forever under quiescence (after some point always returning to the same state), while terminating under saturation.

3.7 Clause Ordering

Saturation checking is implemented by term indexing. Specifically, LolliMon unifies unrestricted and linear hypotheses, distinguished by appropriate tags, in one context, which is implemented as a discrimination tree [26]. The discrimination tree allows the system to efficiently (i.e. without scanning the entire context) check whether a given formula is already in the context, and thus whether a step is non-trivial. However, it also complicates keeping track of clause order.

As specified in Section 3.3, during backchaining the system will try predicate clauses, starting from the most recently assumed, and working back to the least recent. Such behavior, though standard in traditional logic programming languages, is too limiting for some natural LolliMon programs, such as our π-calculus encoding or the meta-circular interpreter.[3] To this end, LolliMon has another directive, `#fair`, which declares the programmer's intent to have the clauses of a particular predicate chosen fairly, rather than in a fixed order. The implementation approximates fairness by trying clauses in a random order.

Including the idea of fair choice is important when we want to be able to simulate a concurrent system in such a way that any given run of the LolliMon program will give us a "typical" execution, rather than some "special" execution. For example, in the π-calculus interpreter described above, the fair-choice model allows both possible outcomes, P and Q, for the process

$$\nu c.\bar{c}\langle c\rangle \mid (c(x).P) \mid (c(x).Q),$$

whereas the fixed-order mode would rule out one or the other of P and Q.

When in the monadic left focusing judgment, any clauses having heads of the form $\{S\}$, rather than an atomic predicate, are eligible to be selected. The implementation always chooses from among these clauses in a fair way. Only for atomic predicates P is the fixed-order mode available.

4. EXAMPLES

In this section we present three concrete examples of LolliMon specifications. We use `typewriter` font for LolliMon code. All of these examples, and many more, are bundled with the prototype distribution of LolliMon.[4]

We summarize LolliMon concrete syntax as follows:

$\multimap$ = `-o`	$\supset$ = `=>`	$\otimes$ = `,`	$\&$ = `&`
$\{\,\}$ = `{ }`	$!$ = `!`	$\top$ = `top`	$\mathbf{1}$ = `one`

Additionally we have

$$\begin{aligned} \lambda x{:}\tau.M &= \texttt{x \textbackslash\ M} \\ \forall x{:}\tau.A &= \texttt{pi x \textbackslash\ A} \\ \exists x{:}\tau.S &= \texttt{sigma x \textbackslash\ S} \end{aligned}$$

where `M`, `A` and `S` are the concrete representations of M, A and S. The type of bound variables `x` is automatically inferred by the system. This is always possible, since all constants are declared and types are restricted to prenex polymorphism.

LolliMon also provides several built-in predicates and terms including lists (`nil` for empty list and `::` for cons), basic integer arithmetic (`is`, `+`, etc.), and output (`write` and `nl`).

We allow clauses of the form `(A, B) -o {C}`, which stands for `B -o (A -o {C})`, and, in a similar manner, `(A, B) => {C}`, which stands for `B => (A => {C})`. Because of the subgoal selection strategy, both of these will first match `A`, then `B`, and then commit the forward chaining step.

We also use the standard convention that (unbound) uppercase letters represent logic variables that are implicitly universally quantified at the outermost level.

4.1 CKY Parsing

We present a LolliMon specification of a CKY parser based on McAllester's logical algorithm [15]. This example only uses the unrestricted context and relies upon forward chaining and saturation.

The main predicate is `parse X I J` which asserts that the input substring from position i to position j, inclusive, is of syntactic category X.

Starting from an input list, we assert `s` i c for each character of input c at position i. When the list becomes empty, we initiate a forward chaining computation, as indicated by the monad brackets in the first clause below. When the forward chaining computation saturates, we test if the whole string (which goes from 1 to n) is of category S and succeed if that is so.

```
load N nil S <= {parse S 1 N}.
load I (C::Cs) S <=
  J is I+1, (s J C => load J Cs S).

start Cs S <= load 0 Cs S.
```

Grammar productions are in normal form of either $X \to c$ for a character c or $X \to Y\ Z$ for non-terminals Y and Z. These are represented in the form `rule X (char C)` and `rule X (jux Y Z)`, respectively. We just run these grammar rules from right to left, for example asserting we have an X from i to k if we already know that the substring from i to j is a Y and the substring from $j+1$ to k is a Z.

[3] Available with the LolliMon distribution.

[4] http://www.cs.cmu.edu/~fp/lollimon/

```
rule X (char C), s I C => {!parse X I I}.

rule X (jux Y Z), parse Y I J, J' is J+1,
  parse Z J' K => {!parse X I K}.
```

This concludes the parser. Note that it does not use the linear context and relies entirely on saturation for termination.

4.2 Call-by-Name and Futures

We present a LolliMon implementation of a λ-calculus with a call-by-name semantics, then add futures, a simple construct for parallel evaluation [10]. This core can be extended to functional languages with features such as recursion, polymorphism, mutable reference, continuations, exceptions, concurrency in the style of Concurrent ML, and distributed computation. Some of these can be found with the prototype distribution of LolliMon or as sample specifications in an earlier report on CLF [4].

Our presentation uses *linear destination-passing style* [23], which is based on three main syntactic categories: *expressions* to be evaluated, *frames* for suspended computations, and *destinations* for values.

Destinations are special in that they are just abstract names and have no further structure. They will be represented as parameters in our implementation.

Corresponding to the three syntactic categories we have three predicates: `eval` evaluates an expression, given a destination, `comp` captures a suspended computation together with the eventual destination of its result, and `return` records the value returned to a destination.

Unlike the usual techniques from (higher-order) logic programming languages, these predicates will take on their appropriate operational meaning in the *linear context*, and we execute using forward chaining with only trivial backward chaining to match atomic formulas. We start with a linear context containing only `eval` e d_0 for an expression e to be evaluated and an initial destination d_0 and reach quiescence in a state with `return` v d_0 for a value v. In this example, values are just expressions, although in some examples it is clearer to separate them out in their own syntactic class.

As a first simple example, consider call-by-name evaluation. We represent expressions using higher-order abstract syntax.

```
lam : (exp -> exp) -> exp.
app : exp -> exp -> exp.
```

A λ-abstraction is a value, so we return it directly to the given destination D.

```
eval (lam x \ E x) D
-o {return (lam x \ E x) D}.
```

The right-hand side of the linear implication is a monadic formula, which enforces that this rule is used only during forward chaining. This will be true for all other rules in this implementation, except for one at the top level.

An application is evaluated by first evaluating the function, while a frame waits for the evaluation's result. The result must be passed in a fresh destination d_1, which is a parameter created by the $\exists_L$ rule.

```
eval (app E1 E2) D
-o {sigma d1 \ eval E1 d1,
       comp (app1 d1 E2) D}.
```

The destination `D` of the new frame is the same as the original destination for the application.

Finally, when evaluation of the function is complete, we can substitute the argument into the function body and evaluate the result. This substitution is modeled by meta-level application, a standard technique with higher-order abstract syntax. Here we need to coordinate *two* linear assumptions: one returns a value, while the other is the waiting frame.

```
return (lam x \ E1' x) D1,
comp (app1 D1 E2) D
-o {eval (E1' E2) D}.
```

This completes the main program. In order to invoke it at the top level, we have a predicate `evaluate` which evaluates an expression e and prints its value. To print the result we use the built-in predicates `write` and `nl`, the latter starting a new line.

```
evaluate E
  o- (pi d0 \ eval E d0
        -o {sigma V \ return V d0,
               write V, nl}).
```

A query `evaluate` e now starts with backward chaining (note the direction `o-` of the outermost implication), creating the initial destination d_0 and linear assumption `eval` e d_0 before entering the monad and initiating forward chaining. We continue to forward chain until we reach quiescence, which is the case precisely when we are returning a value to the initial destination, or the Mini-ML computation gets stuck. This cannot happen for a well-typed Mini-ML term, but we have elided the necessary type-checking predicate here, so it is possible for an `evaluate` query to fail.

As a generalization of the above we consider *futures* derived from Multilisp [10]: the construct `future` e immediately returns with a *promise* and starts a new thread evaluating e in parallel. If the promise is ever evaluated (touched) we block until the evaluation of e completes, which will give us the value of the promise.

The most straightforward implementation in linear destination-passing style uses new expression forms `future` e and `promise` p where p is a destination. We also use a new frame `future1` d_1 to await the completion of the future. Evaluating `future` e then creates two new destinations: one, called d_1, to stand for the value of e directly, and one, called p, to represent the promise.

```
eval (future E) D
-o {sigma d1 \ sigma p \
      eval E d1, comp (future1 d1) p,
      return (promise p) D}.
```

If evaluation of E completes with value V, we install V as the value of the future p. This must occur in the unrestricted context: the promise p may actually be replicated many times via substitution, or it may never be needed at all, violating linearity in both cases.

```
return V D1,
comp (future1 D1) P
-o {!return V P}.
```

If we ever evaluate (touch) a promise, we create a frame waiting for its value.

```
eval (promise P) D
-o {comp (promise1 P) D}.
```

Finally, if the value of the promise is both available and needed, we pass it to the proper destination.

```
!return V1 P,
comp (promise1 P) D
-o {return V1 D}.
```

Thus, futures can be explained in four rules, without affecting the rules for functions and applications previously introduced. Such semantic modularity is a an elegant property of linear destination-passing style, and it is immediately reflected in the LolliMon encoding.

4.3 Checking Graph Bipartiteness

We now present a LolliMon specification of graph bipartiteness checking based on the logical algorithms in the formulation of Ganzinger and McAllester [8].

A graph is represented by a set of unrestricted assumptions `edge` u v for nodes u and v, indicating an edge from u to v. We also start with a *linear* assumption `unlabeled` u for every node u. We further have constant labels `a` and `b` to represent two partitions.

The outer loop of the algorithm picks an arbitrary unlabeled node, labels it (arbitrarily) with `a`, and then propagates information by labeling all neighbors of `a` nodes with `b` and vice versa. When this propagation saturates, we check if there are any nodes labeled both `a` and `b`. If so, the graph cannot be bipartite. If not, we pick another unlabeled node, label it with `a` and repeat the propagation step. If there are no unlabeled nodes left we know the graph is bipartite.

First, the top-level iteration. In order to cut off unnecessary iterations, we check for a node labeled both `a` and `b` first (note use of the `#ordered` directive). So `iterate` succeeds if the graph is *not* bipartite.

```
#ordered iterate.
iterate o- sigma U \
  !labeled U a, !labeled U b, top.
iterate o- sigma U \
  unlabeled U, (labeled U a => {iterate}).
```

By the structure of the second clause, the overall computation consists of a variable number of stages, each of which is a computation run to saturation. Assumptions of the form `labeled` u are always unrestricted, indicated by the form `A => {B}` rather than linear implication `A -o {B}`. `unlabeled` assumptions are always linear, so the second line consumes one. We use `top` in the first clause so we can stop iterating even if there are unlabeled nodes left.

The next three rules are for label propagation, including one to remove linear assumptions `unlabeled U` in case the node `U` was labeled during the saturation process.

```
!labeled U a, !edge U V -o {!labeled V b}.
!labeled U b, !edge U V -o {!labeled V a}.
!labeled U K, unlabeled U -o {one}.
```

Finally we have a rule to generate the symmetric closure of the initial edge relation, again relying on saturation.

```
!edge U V -o {!edge V U}.
```

This completes the implementation. An interesting aspect of it is the use of linearity to avoid explicit deletions. These are present in Ganzinger and McAllester's specification, but have no discernible logical justification.

5. CONCLUSION

We have presented a programming language, LolliMon, based on intuitionistic linear logic augmented with a monad. Computation in LolliMon proceeds via proof construction. In the asynchronous fragment (outside the monad), computation uses backward chaining and backtracking as in the Lolli language [12]. In the synchronous fragment (inside the monad) computation uses forward chaining and committed choice, which allows natural models of concurrency and saturation-based algorithms. The interaction between these strategies is rich, yet the monadic structure of the underlying logic keeps it manageable. We have presented several examples, and more are provided with the implementation.

On the logical side, the first items of future work are to gain a better understanding of additive disjunction ($\oplus$) and falsehood ($\mathbf{0}$), and to extend the operational semantics to the fully dependent type theory CLF [29].

On the computational side, we plan to consider issues of fair scheduling and the difficulties that arise from free variables during forward chaining and, in particular, with respect to saturation. We also plan to give a more thorough analysis of unification and unification constraints.

As far as the implementation is concerned, the most pressing need is to devise techniques for more efficient forward chaining.

Finally, we are interested in developing analysis tools for LolliMon programs, such as mode or termination checkers. This should be particularly interesting for forward chaining, where it may be possible to use model checking for state-space exploration to establish temporal properties of LolliMon program executions, or to support automatic complexity analysis in the style advocated by Ganzinger and McAllester [7].

6. REFERENCES

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

[2] J.-M. Andreoli and R. Pareschi. LO and behold! Concurrent structured processes. In *OOPSLA/ECOOP '90: Proceedings on object-oriented programming systems, languages, and applications*, pages 44–56, Ottawa, Canada, 1990. ACM Press.

[3] Y. Bekkers and P. Tarau. Monadic constructs for logic programming. In J.Lloyd, editor, *Proceedings of the International Symposium on Logic Programming (ILPS'95)*, pages 51–65, Portland, Oregon, Dec. 1995. MIT Press.

[4] I. Cervesato, F. Pfenning, D. Walker, and K. Watkins. A concurrent logical framework II: Examples and applications. Technical Report CMU-CS-02-102, Department of Computer Science, Carnegie Mellon University, 2002. Revised May 2003.

[5] M. Fairtlough and M. Mendler. Propositional lax logic. *Information and Computation*, 137(1):1–33, Aug. 1997.

[6] T. Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37(1–3):95–138, Oct. 1998. Special Issue on Constraint Logic Programming.

[7] H. Ganzinger and D. A. McAllester. A new meta-complexity theorem for bottom-up logic programs. In T. R.Goré, A.Leitsch, editor, *Proceedings of the First International Joint Conference on ArAutomated Reasoning (IJCAR'01)*, pages 514–528, Siena, Italy, June 2001. Springer-Verlag LNCS 2083.

[8] H. Ganzinger and D. A. McAllester. Logical algorithms. In P.Stuckey, editor, *Proceedings of the 18th International Conference on Logic Programming*, pages 209–223, Copenhagen, Denmark, July 2002. Springer-Verlag LNCS 2401.

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

[10] R. H. Halstead. Multilisp: A language for parallel symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–539, Oct. 1985.

[11] J. Harland, D. Pym, and M. Winikoff. Forward and backward chaining in linear logic. In D. Galmiche, editor, *Proceedings of the Workshop on Type-Theoretic Languages: Proof Search and Semantics*, volume 37 of *Electronic Notes in Theoretical Computer Science*, Pittsburgh, Pennsylvania, June 2000. Elsevier Science.

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

[13] N. Kobayashi and A. Yonezawa. Typed higher-order concurrent linear logic programming. Technical Report 94-12, Department of Information Science, University of Tokyo, July 1994.

[14] P. López and J. Polakow. Implementing efficient resource management for linear logic programming. In F. Baader and A. Voronkov, editors, *Eleventh International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR'05)*, pages 528–543, Montevideo, Uruguay, Mar. 2005. Springer-Verlag LNAI 3452.

[15] D. A. McAllester. On the complexity analysis of static analyses. *Journal of the ACM*, 49(4):512–537, 2002.

[16] R. McGrail. *Monads and Control in Logic Programming*. PhD thesis, Wesleyan University, 1997.

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

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

[19] D. Miller and G. Nadathur. Higher-order logic programming. In E. Shapiro, editor, *Proceedings of the Third International Logic Programming Conference*, pages 448–462, London, June 1986.

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

[21] R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[22] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[23] F. Pfenning. Substructural operational semantics and linear destination-passing style. In W.-N. Chin, editor, *Proceedings of the 2nd Asian Symposium on Programming Languages and Systems (APLAS'04)*, page 196, Taipei, Taiwan, Nov. 2004. Springer-Verlag LNCS 3302.

[24] F. Pfenning and R. Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11:511–540, 2001.

[25] D. Pym and J. Harland. A uniform proof-theoretic investigation of linear logic programming. *Journal of Logic and Computation*, 4(2):175–207, April 1994.

[26] I. Ramakrishnan, R. Sekar, and A. Voronkov. Term indexing. In *Handbook of Automated Reasoning, vol. 2*, pages 1853–1965. Elsevier Science and MIT Press, 2001.

[27] E. Y. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):413–510, Sept. 1989.

[28] P. Wadler. The essence of functional programming. In *Conference Record of the 19th Symposium on Principles of Programming Languages*, pages 1–14, Albuquerque, Jan. 1992. ACM Press.

[29] K. Watkins, I. Cervesato, F. Pfenning, and D. Walker. A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-02-101, Department of Computer Science, Carnegie Mellon University, 2002. Revised May 2003.

[30] K. Watkins, I. Cervesato, F. Pfenning, and D. Walker. A concurrent logical framework: The propositional fragment. In S. Berardi, M. Coppo, and F. Damiani, editors, *Types for Proofs and Programs*, pages 355–377. Springer-Verlag LNCS 3085, 2004. Revised selected papers from the *Third International Workshop on Types for Proofs and Programs*, Torino, Italy, April 2003.