Modular Multiset Rewriting

Iliano Cervesato and Edmund S. L. Lam

Carnegie Mellon University
iliano@cmu.edu and sllam@andrew.cmu.edu

Abstract. Rule-based languages are being used for ever more ambitious applications. As program size grows however, so does the overhead of team-based development, reusing components, and just keeping a large flat collection of rules from interfering. In this paper, we propose a module system for a small logically-motivated rule-based language. The resulting modules are nothing more than rewrite rules of a specific form, which are themselves just logic formulas. Yet, they provide some of the same features found in advanced module systems such as that of Standard ML, in particular name space separation, support for abstract data types, and parametrization (functors in ML). Our modules also offer essential features for concurrent programming such as facilities for sharing private names. This approach is directly applicable to other rule-based languages, including most forward-chaining logic programming languages and many process algebras.

1 Introduction

Rule-based programming, a model of computation by which rules modify a global state by concurrently rewriting disjoint portions of it, is having a renaissance as a number of domains are finding a use for its declarative and concise specifications, natural support for concurrency, and relative ease of reasoning [2,8,12,14,15]. Furthermore, the asynchronous state transformation model it embodies has been shown to subsume various models of concurrency [6], in particular multiset rewriting, Petri nets and process algebra [20], and several general-purpose languages based on it have been proposed [3,10].

As languages gain popularity, the need for modularity emerges, since the overhead associated with writing code grows with program size. Modularity tames complexity. In traditional programming languages, it addresses the challenges of breaking a large program into a hierarchy of components with clear interfaces, swappable implementations, team-based development, dependency management, code reuse, and separate compilation. Yet, even rule-based languages used for sizable applications [10,15] provide no support for modular programming.

Programming-in-the-large in a rule-based languages brings about additional challenges not typically found in imperative or functional languages. First, languages such as Datalog [12] and CHR [10] have a flat name space which gives no protections against accidentally reusing a name. Moreover, each rule in them adds to the definition of the names it contains rather than overriding them. Second, these languages tend to have an open scope, meaning that there is no support for local definitions or private names. Finally, a rule can apply as soon as its prerequisites enter the global state, as opposed to when a procedure is called in a conventional language. This, together with the pitfalls of concurrency, makes writing correct code of even moderate size difficult. These

challenges make enriching rule-based languages with a powerful module system all the more urgent if we want them to be used for large applications.

In this paper, we develop a module system for a small rule-based programming language. This language, $\mathcal{L}^1$, subsumes many languages founded on multiset rewriting, forward-chaining proof search and process algebra [6]. Moreover, $\mathcal{L}^1$ is also a syntactic fragment of intuitionistic linear logic in that state transitions map to derivable sequents. In fact, the transition rules for each operator of $\mathcal{L}^1$ correspond exactly to the left sequent rules of this logic, and furthermore the notion of a whole-rule rewriting step originates in a focused presentation of proof search for it [21].

We engineer a module system for $\mathcal{L}^1$ by observing that certain programming patterns capture characteristic features of modularity such as hiding implementation details, providing functionalities to client code through a strictly defined interface, parametricity and the controlled sharing of names. We package these patterns into a handful of constructs that we provide to the programmer as a syntactic extension of $\mathcal{L}^1$ we call $\mathcal{L}^M$. The module system of $\mathcal{L}^M$ supports many of the facilities for modular programming found in Standard ML [19], still considered by many an aspirational gold standard, in particular fine-grained name management and module parametricity (functors). Furthermore, $\mathcal{L}^M$ naturally supports idioms such as higher-order functors and recursive modules, which are not found in [19]. Yet, because the modular constructs of $\mathcal{L}^M$ are just programming templates in $\mathcal{L}^1$, programs in $\mathcal{L}^M$ can be faithfully compiled into $\mathcal{L}^1$. Moreover, since $\mathcal{L}^1$ subsumes the model of computation of a variety of rule-based languages (including those founded on forward-chaining, multiset rewriting and process algebra), it provides a blueprint for enriching these languages with a powerful, yet lightweight and declarative, module system.

With a few exceptions such as [17], research on modularity for rule-based languages has largely targeted backward-chaining logic programming [4]. Popular open-source and commercial implementations of Prolog (e.g., SWI Prolog and SICStus Prolog) do provide facilities for modular programming although not in a declarative fashion. The present work is inspired by several attempts at understanding modularity in richer backward-chaining languages. In particular [18] defines a module system for λProlog on the basis of this language's support for embedded implication, while [1] achieves a form of modularization via a mild form of second-order quantification [11].

The main contributions of this paper are threefold. First, we define a language, $\mathcal{L}^1$, that resides in an operational sweet spot between the stricture of traditional rule-based languages and the freedom of the rewriting reading of intuitionistic linear logic [6]. Second, we engineer a powerful module system on top of this core language with support for name space separation, parametricity, and controlled name sharing. Third, we show that this module infrastructure is little more than syntactic sugar over the core language, and can therefore be compiled away. In fact, this work provides a logical foundation of modularity in rule-based languages in general.

The remainder of this paper is organized as follows: Section 2 defines the language $\mathcal{L}^1$, Section 3 introduces our modular infrastructure through examples, Section 4 collects the resulting language $\mathcal{L}^M$ and elaborates it back into $\mathcal{L}^1$, and Section 5 outlines future developments. Further details can be found in the companion technical report [5].

2 Core Language

This section develops a small, logically-motivated, rule-based language that will act as the core language in which we write (non-modular) programs. It is also the language our modular infrastructure will compile into.

2.1 Multiset Rewriting with Existentials and Nested Rules

Our core formalism, which we call $\mathcal{L}^1$, is a first-order multiset rewriting language extended with dynamic name generation and support for nested rewrite rules. As such, it is a fragment of the logically-derived language of ω-multisets studied in [6]. Because we are interested in writing actual programs, we consider a simply-typed variant.

The syntax of $\mathcal{L}^1$ is specified by the following grammar (the productions on the far right — in blue — will be discussed in Section 2.3):

$$
\begin{array}{rrll}
\textit{Types} & \tau & ::= \iota \mid o \mid \tau \to \tau \mid \tau \times \tau \mid \top & \\
\textit{Terms} & t & ::= x \mid f\ t \mid (t, t) \mid () & \mid X \mid p \\
\textit{Atoms} & A & ::= p\ t & \mid X\ t \\
\textit{LHS} & l & ::= \cdot \mid A, l & \\
\textit{Rules} & R & ::= l \multimap P \mid \forall x : \iota.\, R & \mid \forall X : \tau \to o.\, R \\
\textit{Programs} & P & ::= \cdot \mid P, P \mid A \mid R \mid\ !R \mid \exists x : \tau \to \iota.\, P & \mid \exists X : \tau \to o.\, P
\end{array}
$$

Terms, written t, are built from other terms by pairing and by applying function symbols, f. The starting point is either the unit term $()$ or a term variable, generically written x. In examples, we abbreviate $f\ ()$ to just f. We classify terms by means of simple types, denoted τ. We consider two base types, the type of terms themselves, denoted ι, and the type of formulas, denoted o. Type constructors match term constructors, with the type of $()$ written $\top$. This minimal typing infrastructure can be considerably enriched with additional type and term constructors, as done in some examples of Section 3.

Programs are built out of rules, left-hand sides, and ultimately atoms. An *atom* A is a predicate symbol p applied to a term. A *rule*, R, is the universal closure of a rewrite directive of the form $l \multimap P$. The *left-hand side* l is a multiset of atoms, where we write "$\cdot$" for the empty multiset and "A, l" for the extension of l with atom A. We consider "," commutative and associative with unit "$\cdot$". The right-hand side P of a rewrite directive is a multiset of either atoms A, single-use rules R or reusable rules $!R$. A right-hand side can also have the form $\exists x : \tau \to \iota.\, P$, which, when executed, will have the effect of creating a new function symbol for x of type $\tau \to \iota$ for use in P. As rules consist of a rewrite directive embedded within a layer of universal quantifiers, generically $\forall x_1 : \tau_1. \ldots \forall x_n : \tau_n. (l \multimap P)$ with τ_i equal to ι for the time being, we will occasionally use the notation $\forall \vec{x} : \vec{\tau}. (l \multimap P)$ where $\vec{x}$ stands for $x_1, \ldots, x_n$ and $\vec{\tau}$ for $\tau_1, \ldots, \tau_n$. A *program* is what we just referred to as a right-hand side. A program is therefore a collection of atoms, single-use and reusable rules, and existentially quantified programs.

The quantifiers $\forall x : \tau.\, R$ and $\exists x : \tau.\, P$ are binders in $\mathcal{L}^1$. We adopt the standard definitions of free and bound variables, closed expressions (called ground in the case of terms and atoms), and α-renaming. Given a syntactic entity O possibly containing a free variable x, we write $[t/x]O$ for the capture-avoiding substitution of term t of the

same type for every free occurrence of x in O. Given sequences $\vec{x}$ and $\vec{t}$ of variables and terms of the same length, we denote the simultaneous substitution of every term t_i in $\vec{t}$ for the corresponding variable x_i in $\vec{x}$ in O as $[\vec{t}/\vec{x}]O$. We write θ for a generic substitution $\vec{t}/\vec{x}$ and $O\theta$ for its application.

Function and predicate symbols have types of the form $\tau \to \iota$ and $\tau \to o$ respectively. The symbols in use during execution together with their type are collected in a *signature*, denoted Σ. We treat free existential variables as symbols and account for them in the signature. The type of free universal variables are collected in a *context* Γ. Signatures and contexts are defined as follows (again, ignore the rightmost production):

$$\begin{array}{rrcl} \textit{Signatures} & \Sigma & ::= & \cdot \mid \Sigma, f : \tau \to \iota \mid \Sigma, p : \tau \to o \\ \textit{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : \iota \qquad\qquad \mid \Gamma, X : \tau \to o \end{array}$$

We write $\Gamma \vdash_\Sigma t : \tau$ to mean that term t has type τ in Γ and Σ. The routine typing rules for this judgment can be found in [5], together with the straightforward definition of validity for the other entities of the language. Valid rewrite directives $l \multimap P$ are also subject to the *safety requirement* that the free variables in P shall occur in l or in the left-hand side of an enclosing rule.

Computation in $\mathcal{L}^1$ takes the form of state transitions. A *state* is a pair $\Sigma.\, \Pi$ consisting of a closed program Π and a signature Σ that accounts for all the function and predicate symbols in Π. We emphasize that the program must be closed by writing it as Π rather than P. Since rules R in Π are closed, we further abbreviate $\forall \vec{x} : \vec{\tau}.\, (l \multimap P)$ as $\forall (l \multimap P)$. Observe that a state contains both atoms, which carry data, and rules, which perform computation on such data.

We give two characterizations of the rewriting semantics of $\mathcal{L}^1$, each rooted in the proof theory of linear logic as discussed in Section 2.2. The *unfocused rewriting semantics* interprets each operator in $\mathcal{L}^1$ as an independent state transformation directive. It is expressed by means of a step judgment of the form

$$\Sigma.\, \Pi \mapsto \Sigma'.\, \Pi' \qquad \textit{State } \Sigma.\, \Pi \textit{ transitions to state } \Sigma'.\, \Pi' \textit{ in one step}$$

In this semantics, each operator in the language is understood as a directive to carry out one step of computation in the current state. Therefore, each operator yields one transition rule, given in the following table (please, ignore the starred entries in blue):

$$\begin{array}{rcll} \Sigma.\, (\Pi, l, l \multimap P) & \mapsto & \Sigma.\, (\Pi, P) & \\ \Sigma.\, (\Pi, \forall x : \iota.\, R) & \mapsto & \Sigma.\, (\Pi, [t/x]R) & \text{if } \cdot \vdash_\Sigma t : \iota \\ \Sigma.\, (\Pi, \forall X : \tau \to o.\, R) & \mapsto & \Sigma.\, [p/X]R & \text{if } p : \tau \to o \text{ in } \Sigma \quad (*) \\ \Sigma.\, (\Pi, !R) & \mapsto & \Sigma.\, (\Pi, !R, R) & \\ \Sigma.\, (\Pi, \exists x : \tau \to \iota.\, P) & \mapsto & (\Sigma, x : \tau \to \iota).\, (\Pi, P) & \\ \Sigma.\, (\Pi, \exists X : \tau \to o.\, P) & \mapsto & (\Sigma, X : \tau \to o).\, \Pi, P & (*) \end{array}$$

In words, $\multimap$ is a rewrite directive which has the effect of identifying its left-hand side atoms in the surrounding state and replacing them with the program in its right-hand side. The operator $\forall$ is an instantiation directive: it picks a term of the appropriate type and replaces the bound variable with it. Instead, ! is a replication directive, enabling

a reusable rule to be applied while keeping the master copy around. Finally, $\exists$ is a name generation directive which installs a new symbol of the appropriate type in the signature.

The rules of this semantics are pleasantly simple as they tease out the specific behavior of each individual language construct. However, by considering each operator in isolation, this semantics falls short of the expected rewriting behavior. Consider the state $\Sigma.\, \Pi = (p : \iota \to o, q : \iota \to o, a : \iota, b : \iota).\, (p\ a, \forall x : \iota.\, p\ x \multimap q\ x)$. From it, the one transition sequence of interest is $\Pi \mapsto p\ a, (p\ a \multimap q\ a) \mapsto q\ a$, where we have omitted the signature Σ for succinctness. However, nothing prevents picking the "wrong" instance of x and taking the step $\Pi \mapsto p\ a, (p\ b \multimap q\ b)$ from where we cannot proceed further. This second possibility is unsatisfactory as it does not apply the rule fully.

The *focused rewriting semantics* makes sure that rules are either fully applied, or not applied at all. It corresponds to the standard operational semantics of most languages based on multiset rewriting, forward chaining and process transformation. It also leverages the observation that some of the state transformations associated with individual operators, here the existential quantifier, never preempt other transitions from taking place, while others do (here both the universal instantiation and the rewrite directive).

A closed program without top-level existential quantifiers is called *stable*. We write $\boldsymbol{\Pi}$ for a state program Π that is stable. A stable state has the form $\Sigma.\, \boldsymbol{\Pi}$. The focused operational semantics is expressed by two judgments:

$\Sigma.\, \Pi \Mapsto \Sigma'.\, \Pi'$	*Non-stable state* $\Sigma.\, \Pi$ *transitions to state* $\Sigma'.\, \Pi'$ *in one step*
$\Sigma.\, \boldsymbol{\Pi} \Mapsto \Sigma'.\, \Pi'$	*Stable state* $\Sigma.\, \boldsymbol{\Pi}$ *transitions to state* $\Sigma'.\, \Pi'$ *in one step*

The first judgment is realized by selecting an existential program component in Π and eliminating the quantifier by creating a new symbol. A finite iteration yields a stable state. At this point, the second judgment kicks in. It selects a rule and fully applies it. To fully apply a rule $\forall(l \multimap P)$, the surrounding state must contain an instance $l\theta$ of the left-hand side l. The focused semantics replaces it with the corresponding instance $P\theta$ of the right-hand side P. The resulting state may not be stable as $P\theta$ could contain existential components. The following (non-starred) transitions formalize this insight.

$$\begin{array}{ll} \Sigma.\, (\Pi, \exists x : \tau \to \iota.\, P) \Mapsto (\Sigma, x : \tau \to \iota).\, (\Pi, P) & \\ \Sigma.\, (\Pi, \exists X : \tau \to o.\, P) \Mapsto (\Sigma, X : \tau \to o).\, (\Pi, P) & (*) \\ \Sigma.\, \boldsymbol{\Pi, l\theta, \forall(l \multimap P)} \Mapsto \Sigma.\, (\boldsymbol{\Pi}, P\theta) & \\ \Sigma.\, \boldsymbol{\Pi, l\theta, !\forall(l \multimap P)} \Mapsto \Sigma.\, (\boldsymbol{\Pi, !\forall(l \multimap P)}, P\theta) & \end{array}$$

Both the focused and unfocused semantics transform valid states into valid states [5]. Furthermore, any transition step achievable in the focused semantics is also achievable in the unfocused semantics, although possibly in more than one step in the case of rules.

Theorem 1.

1. *If* $\Sigma.\, \Pi \Mapsto \Sigma'.\, \Pi'$, *then* $\Sigma.\, \Pi \mapsto \Sigma'.\, \Pi'$.
2. *If* $\Sigma.\, \boldsymbol{\Pi} \Mapsto \Sigma'.\, \Pi'$, *then* $\Sigma.\, \boldsymbol{\Pi} \mapsto^* \Sigma'.\, \Pi'$.

where $\mapsto^*$ is the reflexive and transitive closure of $\mapsto$. This property is proved in [5]. The reverse does not hold, as we saw earlier.

The language $\mathcal{L}^1$ is a syntactic fragment of the formalism of ω-multisets examined in [6] as a logical reconstruction of multiset rewriting and some forms of process algebra. The main restriction concerns the left-hand side of rules, which in $\mathcal{L}^1$ is a multiset of atoms, while in an ω-multiset it can be any formula in the language. This restriction makes implementing rule application in $\mathcal{L}^1$ much easier than in the general language, is in line with all rule-based languages we are aware of, and is endorsed by a focusing view of proof search. Therefore, $\mathcal{L}^1$ occupies a sweet spot between the free-wheeling generality of ω-multiset rewriting and the implementation simplicity of many rule-based languages. $\mathcal{L}^1$ also limits the usage of the ! operator to just rules. This avoids expressions that are of little use in programming practice (for example doubly reusable rules $!!R$ or left-hand side atoms of the form $!A$). This is relaxed somewhat in [5] where we allow reusable atoms in the right-hand side of a rule. We also left out the choice operator of ω-multisets (written $\&$) because we did not need it in any of the examples in this paper. Adding it back is straightforward.

Syntactic fragments of $\mathcal{L}^1$ correspond to various rule-based formalisms. First-order multiset rewriting, as found for example in CHR [10], relies on reusable rules whose right-hand side is a multiset of atoms, and therefore corresponds to $\mathcal{L}^1$ rules of the form $!\forall\vec{x}.\,(l_1 \multimap l_2)$. Languages such as MSR additionally permit the creation of new symbols in the right-hand side of a rule, which is supported by $\mathcal{L}^1$ rules of the form $!\forall\vec{x}.\,(l_1 \multimap \exists\vec{y}.\,l_2)$.

As shown in [6], ω-multiset rewriting, and therefore $\mathcal{L}^1$, also subsumes many formalisms based on process algebra. Key to doing so is the possibility to nest rules (thereby directly supporting the ability to sequentialize actions), a facility to create new symbols (which matches channel restriction), and the fact that multisets are commutative monoids (like processes under parallel composition). For example, the asynchronous π-calculus [20] is the fragment of $\mathcal{L}^1$ where rule left-hand sides consist of exactly one atom (corresponding to a receive action — send actions correspond to atoms on the right-hand side of a rule).

Our language, like ω-multiset rewriting itself, contains fragments that correspond to both the state transition approach to specifying concurrent computations (as multiset rewriting for example) and specifications in the process-algebraic style. It in fact supports hybrid specifications as well, as found in the Join calculus [9] and in CLF [22].

2.2 Logical Foundations

The language $\mathcal{L}^1$, like ω-multisets [6], corresponds exactly to a fragment of intuitionistic linear logic [13]. In fact, not only can we recognize the constructs of our language among the operators of this logic, but $\mathcal{L}^1$'s rewriting semantics stem directly from its proof theory.

The operators "," and "$\cdot$", $\multimap$, !, $\forall$ and $\exists$ of $\mathcal{L}^1$ correspond to the logical constructs $\otimes$, $\mathbf{1}$, $\multimap$, !, $\forall$ and $\exists$, respectively, of multiplicative-exponential intuitionistic linear logic (MEILL). We write the derivability judgment of MEILL as $\Gamma; \Delta \longrightarrow_\Sigma \varphi$, where φ is a formula, the linear context Δ is a multiset of formulas that can be used exactly once in a proof of φ, while the formulas in the persistent context Γ can be used arbitrarily many times, and Σ is a signature defined as for $\mathcal{L}^1$.

The transitions of the unfocused rewriting semantics of $\mathcal{L}^1$ can be read off directly from the left sequent rules of the above connectives. Consider for example the transition for an existential $\mathcal{L}^1$ program and the left sequent rule $\exists\mathsf{L}$ for the existential quantifier:

$$\Sigma.\,(\Pi, \exists x : \tau \to \iota.\, P) \mapsto (\Sigma, x : \tau \to \iota).\,(\Pi, P) \quad \leftrightsquigarrow \quad \frac{\Gamma; \Delta, \varphi \longrightarrow_{\Sigma, x:\tau\to\iota} \psi}{\Gamma; \Delta, \exists x : \tau \to \iota.\, \varphi \longrightarrow_{\Sigma} \psi}\ \exists\mathsf{L}$$

The antecedent $\Pi, \exists x : \tau \to \iota.\, P$ of the transition corresponds to the linear context $\Delta, \exists x : \tau \to \iota.\, \varphi$ of the rule conclusion, while its consequent (Π, P) matches the linear context of the premise (Δ, φ) and the signatures have been updated in the same way. A similar correspondence applies in all cases, once we account for shared structural properties of states and sequents. In particular, multiplicative conjunction $\otimes$ and its unit $\mathbf{1}$ are the materialization of the formation operators for the linear context of a sequent, context union and the empty context. This means that linear contexts can be interpreted as the multiplicative conjunction of their formulas. There is a similar interplay between persistent formulas $!\varphi$ and the persistent context Γ [6]. Altogether, this correspondence is captured by the following property, proved in [5,6], where $\ulcorner \Pi \urcorner$ is the MEILL formula corresponding to state Π and $\exists \Sigma'.\ulcorner \Pi' \urcorner$ is obtained by prefixing $\ulcorner \Pi' \urcorner$ with an existential quantification for each declaration in the signature Σ'.

Theorem 2. *If $\Sigma.\,\Pi$ is valid and $\Sigma.\,\Pi \mapsto \Sigma'.\,\Pi'$, then $\cdot; \ulcorner \Pi \urcorner \longrightarrow_{\Sigma} \exists \Sigma'.\ulcorner \Pi' \urcorner$.*

The transitions of the focused rewriting semantics of $\mathcal{L}^1$ originate from the focused presentation of linear logic [16], and specifically of MEILL. Focusing is a proof search strategy that alternates two phases: an inversion phase where invertible sequent rules are applied exhaustively and a chaining phase where it selects a formula (the focus) and decomposes it maximally using non-invertible rules. Focusing is complete for many logics of interest, in particular for traditional intuitionistic logic [21] and for linear logic [16], and specifically for MEILL [5]. Transitions of the form $\Sigma.\,\Pi \Mapsto \Sigma'.\,\Pi'$ correspond to invertible rules and are handled as in the unfocused case. Instead, transitions of the form $\Sigma.\,\boldsymbol{\Pi} \Mapsto \Sigma'.\,\Pi'$ correspond to the selection of a focus formula and the consequent chaining phase. Consider for example the transition for a single-use rule $\forall(l \multimap P)$. The derivation snippet below selects a context formula $\forall(\varphi_l \multimap \varphi_P)$ of the corresponding form from a sequent where no invertible rule is applicable (generically written $\Gamma; \boldsymbol{\Delta} \Longrightarrow_{\Sigma} \psi$), puts it into focus (indicated as a red box), and then applies non-invertible rules to it exhaustively. In the transcription of $\mathcal{L}^1$ into MEILL, the formula $\varphi_l\theta$ is a conjunction of atomic formulas matching $l\theta$, which allows continuing the chaining phase in the left premise of $\multimap\mathsf{L}$ into a complete derivation when Δ_2 consists of those exact atoms. The translation φ_P of the program P re-enables an invertible rule and therefore the derivation loses focus in the rightmost open premise.

$$\Sigma.\,\boldsymbol{\Pi}, \boldsymbol{l\theta}, \forall(\boldsymbol{l} \multimap \boldsymbol{P}) \Mapsto \Sigma.\,(\boldsymbol{\Pi}, P\theta) \quad \leftrightsquigarrow \quad \dfrac{\dfrac{\dfrac{\cdots\ \overline{\Gamma; \boldsymbol{A\theta} \Longrightarrow_{\Sigma} \boxed{A\theta}}\ \cdots \quad \dfrac{\Gamma; \boldsymbol{\Delta_1}, \varphi_P\theta \Longrightarrow_{\Sigma} \psi}{\Gamma; \boldsymbol{\Delta_1}, \boxed{\varphi_P\theta} \Longrightarrow_{\Sigma} \psi}\ \mathsf{blurL}}{\Gamma; \boldsymbol{\Delta_2} \Longrightarrow_{\Sigma} \boxed{\varphi_l\theta} \qquad \Gamma; \boldsymbol{\Delta_1}, \boldsymbol{\Delta_2}, \boxed{\varphi_l\theta \multimap \varphi_P\theta} \Longrightarrow_{\Sigma} \psi}\ \multimap\mathsf{L}}{\Gamma; \boldsymbol{\Delta_1}, \boldsymbol{\Delta_2}, \boxed{\forall(\varphi_l \multimap \varphi_P)} \Longrightarrow_{\Sigma} \psi}\ \forall\mathsf{L}(\mathsf{repeated})}{\Gamma; \boldsymbol{\Delta_1}, \boldsymbol{\Delta_2}, \forall(\boldsymbol{\varphi_l} \multimap \boldsymbol{\varphi_P}) \Longrightarrow_{\Sigma} \psi}\ \mathsf{focusL}$$

Reusable $\mathcal{L}^1$ rules $!\forall(l \multimap P)$ are treated similarly. The correspondence is formalized in the following theorem, proved in [5].

Theorem 3. *Let $\Sigma.\, \Pi$ state be a valid state.*

1. *If $\boldsymbol{\Pi}$ is stable and $\Sigma.\, \boldsymbol{\Pi} \Mapsto \Sigma'.\, \Pi'$, then $\cdot; \ulcorner \boldsymbol{\Pi} \urcorner \Longrightarrow_{\Sigma} \exists \Sigma'.\, \ulcorner \Pi' \urcorner$.*
2. *If Π is not stable and $\Sigma.\, \Pi \Mapsto \Sigma'.\, \Pi'$, then $\cdot; \ulcorner \Pi \urcorner \Longrightarrow_{\Sigma} \exists \Sigma'.\, \ulcorner \Pi' \urcorner$.*

In focused logics, the formula patterns involved in a chaining phase can be viewed as *synthetic connectives* and are characterized by well-behaved derived rules. The generic single-use rules $\forall(l \multimap P)$ and reusable rules $!\forall(l \multimap P)$ define such synthetic connectives, and their transitions match exactly the derived left sequent rule in the focused presentation of MEILL.

2.3 Mild Higher-Order Quantification

As we prepare to modularize $\mathcal{L}^1$ programs, it is convenient to give this language a minor second-order flavor. Doing so significantly improves code readability with only cosmetic changes to the underlying formalism and its logical interpretation. In fact, programs in this extended language, that we call $\mathcal{L}^{1.5}$, are readily transformed into $\mathcal{L}^1$ programs, as shown in [5].

The language $\mathcal{L}^{1.5}$ is obtained by taking into account the snippets written in blue throughout the earlier definitions. It extends $\mathcal{L}^1$ with second-order variables and a very weak form of quantification over them. Second-order variables, written X, allow us to define atoms of the form $X\ t$ that are parametric in their predicate. However, we permit these variables to be replaced only with predicate names, which are now legal ingredients in terms. The second-order universal quantifier $\forall X$ carries out this substitution by drawing an appropriate predicate name from the signature. The second-order existential quantifier $\exists X$ extends the signature with new predicate names. This use of second-order entities is closely related to Gabbay and Mathijssen's "one-and-a-halfth-order logic" [11]. It comes short of the expressiveness (and complexity) of a traditional second-order infrastructure (which in our case would permit instantiating a variable X with a parametric program rather than just a predicate name).

The notions of free and bound variables, ground atoms, closed rules and programs, and substitution carry over from the first-order case, and so does the safety requirement. Second-order universal variables are subject to an additional requirement: if a parametric atom $X\,t$ with X universally quantified occurs in the left-hand side of a rule, then X must occur in a term position either in the same left-hand side or in an outer left-hand side. This additional constraint is motivated by the need to avoid rules such as

$$!\forall X : \tau \to o.\, \forall x : \tau.\quad X\ x \multimap \cdot$$

which would blindly delete any atom $p\ t$ for all p of type $\tau \to o$. Instead, we want a rule to be applicable only to atoms it "knows" something about, by requiring that their predicate name be passed to it in a "known" atom. The following rule is instead acceptable

$$!\forall X : \tau \to o.\, \forall x : \tau.\quad \mathsf{delete\_all}(X), X\ x \multimap \cdot$$

as only the predicate names marked to be deleted through the predicate delete_all can trigger this rule. A type system that enforces this restriction is given in [5].

Both the focused and unfocused rewriting semantics of $\mathcal{L}^1$ are extended to process the second-order quantifiers. The notion of state remains unchanged. The properties seen for $\mathcal{L}^1$, in particular the preservation of state validity, also hold for $\mathcal{L}^{1.5}$.

3 Adding Modularity

In this section, we synthesize a module system for $\mathcal{L}^{1.5}$ by examining a number of examples. In each case, we will write rules in a way as to emulate characteristics of modularity, and then develop syntax to abstract the resulting linguistic pattern. For readability, types will be grayed out throughout this section. In most cases, they can be automatically inferred from the way the variables they annotate are used.

Name Space Separation Consider the problem of adding two unary numbers — we write z and $\mathsf{s}(n)$ for zero and the successor of n, respectively, and refer to the type of such numerals as nat. Addition is then completely defined by the single rule

$$!\forall x\colon \mathsf{nat}.\, \forall y\colon \mathsf{nat}.\quad \mathsf{add}(\mathsf{s}(x), y) \multimap \mathsf{add}(x, \mathsf{s}(y))$$

For any concrete values m and n, inserting the atom $\mathsf{add}(m, n)$ in the state triggers a sequence of applications of this rule that will end in an atom of the form $\mathsf{add}(\mathsf{z}, r)$, with r the result of adding m and n. The way this adding functionality will typically be used is by having one rule generate the request in its right-hand side and another retrieve the result in its left-hand side, as in the following client code snippet:

$$\begin{array}{r} \forall m\colon \mathsf{nat}.\, \forall n\colon \mathsf{nat}.\quad \cdots \quad \multimap \cdots, \mathsf{add}(n, m) \\ \forall r\colon \mathsf{nat}.\ \mathsf{add}(\mathsf{z}, r), \cdots \multimap \cdots \end{array}$$

This code is however incorrect whenever there is the possibility of two clients performing an addition at the same time. In fact, any concurrent use of add will cause an interference as the clients have no way to sort out which result (r in $\mathsf{add}(\mathsf{z}, r)$) is who's.

We obviate to this problem by using a second-order existential quantifier to generate the predicate used to carry out the addition, as in the following code snippet where we nested the rule that uses the result inside the rule that requests the addition so that they share the name add.

$$\begin{array}{l} \forall m\colon \mathsf{nat}.\, \forall n\colon \mathsf{nat}.\ \cdots \multimap \exists add\colon \mathsf{nat} \times \mathsf{nat} \to o. \\ \qquad \left[\begin{array}{l} !\forall x\colon \mathsf{nat}.\, \forall y\colon \mathsf{nat}.\ \ add(\mathsf{s}(x), y) \ \multimap add(x, \mathsf{s}(y)), \\ \cdots, add(n, m), \\ \qquad \forall r\colon \mathsf{nat}.\ add(\mathsf{z}, r), \cdots \multimap \cdots \end{array}\right] \end{array}$$

The safety constraint on second-order variables prevents a rogue programmer from intercepting the freshly generated predicate out of thin air. In fact, the rule

$$\forall X\colon \mathsf{nat} \times \mathsf{nat} \to o.\, \forall m\colon \mathsf{nat}.\, \forall n\colon \mathsf{nat}.\quad X(m, n) \multimap X(\mathsf{z}, \mathsf{s}(\mathsf{z}))$$

is invalid as X does not appear in a term position in the left-hand side.

While the above approach eliminates the possibility of interferences, it forces every rule that relies on addition to embed its definition. This goes against the spirit of modularity as it prevents code reuse, reduces maintainability, and diminishes readability.

We address this issue by providing a public name for the addition functionality, for example through the predicate adder, but pass to it the name of the private predicate used by the client code (add). Each client can then generate a fresh name for it. The definition of addition, triggered by a call to adder, can then be factored out from the client code. The resulting rules are as follows:

$$!\forall add\colon \mathsf{nat} \times \mathsf{nat} \to o.\ \mathsf{adder}(add) \multimap \left[\, !\forall x\colon \mathsf{nat}.\, \forall y\colon \mathsf{nat}.\ add(\mathsf{s}(x), y) \multimap add(x, \mathsf{s}(y)) \,\right]$$
$$\forall m\colon \mathsf{nat}.\, \forall n\colon \mathsf{nat}.\ \cdots \multimap \exists add\colon \mathsf{nat} \times \mathsf{nat} \to o.\ \begin{bmatrix} \mathsf{adder}(add), \\ \cdots, add(n, m), \\ \forall r\colon \mathsf{nat}.\ add(\mathsf{z}, r), \cdots \multimap \cdots \end{bmatrix}$$

Observe that, as before, the client generates a fresh name for its private adding predicate ($\exists add\colon \mathsf{nat} \times \mathsf{nat} \to o.$). It now passes it to adder, which has the effect of instantiating the rule for addition with the private name add. The client can then retrieve the result by having $add(\mathsf{z}, r)$ in the left-hand side of an embedded rule like before.

This idea will be the cornerstone of our approach to adding modules to $\mathcal{L}^{1.5}$. We isolate it in the following derived syntax for the exported module (first rule above):

module adder
  provide $add : \mathsf{nat} \times \mathsf{nat} \to o$
  $!\forall m\colon \mathsf{nat}.\, \forall n\colon \mathsf{nat}.\ add(\mathsf{s}(m), n) \multimap add(m, \mathsf{s}(n))$
end

Here, the public name adder is used as the name of the module. The names of the exported operations, the module's interface, are introduced by the keyword **provide**. By isolating the public predicate name adder in a special position (after the keyword **module**), we can statically preempt one newly-introduced problem with the above definition: that a rogue client learn the private name through the atom $\mathsf{adder}(X)$. We shall disallow predicates used as module names (here adder) from appearing in the left-hand side of other rules.

We also provide derived syntax for the client code to avoid the tedium of creating a fresh predicate name and using it properly. Our client is re-expressed as:

$$\forall m\colon \mathsf{nat}.\, \forall n\colon \mathsf{nat}.\ \cdots \multimap A\ \mathbf{as}\ \mathsf{adder}.\ \begin{bmatrix} \cdots, A.add(n, m), \\ \forall r\colon \mathsf{nat}.\ A.add(\mathsf{z}, r), \cdots \multimap \cdots \end{bmatrix}$$

Here, A is a *module reference name* introduced by the line "A **as** adder. ". This syntactic artifact binds A in the right-hand side of the rule. The name A allows constructing *compound predicate names*, here $A.add$, which permits using the exact same names exported by the **provide** stanza of a module. Uses of compound names and the "**as**" construct are elaborated into our original code, as described in Section 4.

Modes In the face of it, our adder module is peculiar in that it requires client code to use an atom of the form $add(\mathsf{z}, r)$ to retrieve the result r of an addition. A better approach is to split add into a predicate add_req for issuing an addition request and a predicate add_res for retrieving the result. However, uses of add_req in the left-hand side of client rules would intercept requests, while occurrences of add_res on their right-hand side could inject forged results. We prevent such misuses by augmenting the syntax of modules with *modes* that describe how exported predicates are to be used. The mode **in** in the **provide** stanza forces a (compound) predicate to be used only on the left-hand side of a client rule, while the mode **out** enables it on the right-hand side only. A declaration without either, for example add earlier, can be used freely. The updated adder module is written as follows in the concrete syntax:

$$
\begin{array}{l}
\textbf{module}\ \mathsf{adder}' \\
\quad \textbf{provide}\ \ \textbf{out}\ add\_req : \mathsf{nat} \times \mathsf{nat} \to o \quad \textbf{in}\ add\_res : \mathsf{nat} \to o \\
\quad\ !\forall x{:}\,\mathsf{nat}.\, \forall y{:}\,\mathsf{nat}.\ \ add\_req(\mathsf{s}(x), y) \multimap add\_req(x, \mathsf{s}(y)) \\
\qquad\qquad\quad \forall z{:}\,\mathsf{nat}.\quad add\_req(\mathsf{z}, z) \multimap add\_res(z) \\
\textbf{end}
\end{array}
$$

and the client code assumes the following concrete form:

$$
\forall m{:}\,\mathsf{nat}.\, \forall n{:}\,\mathsf{nat}.\ \cdots \multimap A\ \textbf{as}\ \mathsf{adder}'.\ \begin{bmatrix} \cdots, A.add\_req(n, m), \\ \forall r{:}\,\mathsf{nat}.\ A.add\_res(r), \cdots \multimap \cdots \end{bmatrix}
$$

Abstract Data Types The infrastructure we developed so far already allows us to write modules that implement abstract data types. Our next module defines a queue with two operations: enqueuing a data item (for us a nat) and dequeuing an item, which we implement as a pair of request and result predicates. An empty queue is created when a queue module is first invoked. We give a name, here QUEUE, to the exported declarations as follows

$$
\begin{array}{l}
\textbf{interface}\ \mathsf{QUEUE} \\
\quad \textbf{out}\ \ enq : \mathsf{nat} \to o \\
\quad \textbf{out}\ deq\_req : o \qquad \textbf{in}\ deq : \mathsf{nat} \to o \\
\textbf{end}
\end{array}
$$

The module queue uses linked lists built from the local predicates $head(d'')$, $tail(d')$ and $data(n, d', d'')$, where the terms d' and d'' identify the predecessor and the successor element, respectively. Such a list will start with atom $head(d_0)$, continue as series of atoms $data(n_i, d_i, d_{i+1})$ and end with the atom $tail(d_n)$.

$$
\begin{array}{l}
\textbf{module}\ \mathsf{queue} \\
\quad \textbf{provide}\ \ \mathsf{QUEUE} \\
\quad \textbf{local}\ \ head{:}\,\iota \to o \quad tail{:}\,\iota \to o \quad data{:}\,\mathsf{nat} \times \iota \times \iota \to o \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \cdot \qquad\qquad\quad \multimap \exists d{:}\,\iota.\, head(d), tail(d) \\
\qquad\quad\ !\forall e{:}\,\mathsf{nat}.\, \forall d{:}\,\iota. \qquad enq(e), head(d) \qquad\quad \multimap \exists d'{:}\,\iota.\, data(e, d', d), head(d') \\
\quad\ !\forall e{:}\,\mathsf{nat}.\, \forall d{:}\,\iota.\, \forall d'{:}\,\iota.\ \ deq\_req, tail(d'), data(e, d, d') \multimap deq(e), tail(d) \\
\textbf{end}
\end{array}
$$

A second implementation based on a term representation of lists can be found in [5].

Sharing Private Names As our next example, we define a module that provides the functionalities of a reference cell in a stateful language. The initial value of the cell will be a parameter v to the module itself, which will be an actual value when the module is instantiated. This corresponds to a functor that takes a value as an argument in Standard ML. This form of parametricity is achieved in $\mathcal{L}^{1.5}$ by simply passing v as an argument to the public predicate used to call the module, in addition to the normal interface predicates [5]. In the concrete syntax, we simply supply v as an argument to the module declaration, as in the following code snippet:

$$
\begin{array}{l}
\textbf{module}\ \mathsf{cell}\ (v\colon \mathsf{nat}) \\
\quad \textbf{provide}\ \ \textbf{out}\ \mathit{get}\ \colon o \qquad\quad \textbf{in}\ \mathit{got}\ \colon \mathsf{nat} \to o \\
\qquad\qquad\ \ \textbf{out}\ \mathit{set}\ \colon \mathsf{nat} \to o \\
\quad \textbf{local}\ \ \mathit{content}\ \colon \mathsf{nat} \to o \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \cdot \qquad\quad \multimap \mathit{content}(v) \\
\qquad\qquad\qquad\qquad\qquad\quad !\forall v\colon \mathsf{nat}. \quad \mathit{get}, \mathit{content}(v) \quad \multimap \mathit{got}(v), \mathit{content}(v) \\
\qquad\qquad\qquad\quad !\forall v\colon \mathsf{nat}.\, \forall v'\colon \mathsf{nat}.\ \ \mathit{set}(v'), \mathit{content}(v) \multimap \mathit{content}(v') \\
\textbf{end}
\end{array}
$$

Reference cells are often shared by various subcomputations. One way to do so is to pass the exported predicates to the subcomputations, after instantiation. In the following example, the client code (first rule) creates a cell C initialized with the value $\mathsf{s}(\mathsf{z})$. It then passes the setter to the second rule through the predicate p and passes the getters to the third rule through the predicate q. The second rule can only write to the cell. The third rule can only read the content of the cell (here, it then outputs it through predicate s).

$$
\begin{array}{lll}
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \cdot & \multimap C\ \textbf{as}\ \mathsf{cell}(\mathsf{s}(\mathsf{z}))\mathsf{.}\ \begin{bmatrix} p(C.\mathit{set}), \\ q(C.\mathit{get}, C.\mathit{got}) \end{bmatrix} \\
\forall \mathit{write}\colon \mathsf{nat} \to o. \qquad p(\mathit{write}) & \multimap \mathit{write}(\mathsf{z}) \\
\begin{bmatrix} \forall \mathit{read\_req}\colon o. \\ \forall \mathit{read}\colon \mathsf{nat} \to o. \end{bmatrix}\ q(\mathit{read\_req}, \mathit{read}) & \multimap \begin{bmatrix} \mathit{read\_req}, \\ \forall r\colon \mathsf{nat}.\ \ \mathit{read}(r) \multimap s(r) \end{bmatrix}
\end{array}
$$

This example shows how the private names obtained through a module invocation can be shared with other rules. Furthermore this sharing can be selective.

Reference cells can be implemented in a much simpler way by exporting just the predicate $\mathit{content}$ that holds the content of the cell [5]. By doing so, however, we prevent the client code from passing distinct capabilities to its subcomputations.

Parametric Modules As our last example, we define a producer-consumer module. We rely on a queue, as defined earlier, to act as the buffer where the producer deposits data and the consumer retrieves them from.

Now, rather than selecting a priori which implementation of queues to use, we make the producer-consumer module parametric with respect to the implementation of queues. This corresponds a functor parametrized by a structure in Standard ML.

$$\begin{array}{l}
\textbf{module}\ \mathsf{prodcons}(Q\colon \mathsf{QUEUE}) \\
\quad \textbf{provide}\ \ \textbf{in}\ \ \ \ \mathit{produce} : \mathsf{nat} \to o \\
\qquad\qquad\ \ \textbf{in}\ \mathit{consume_req} : o \qquad\quad \textbf{out}\ \mathit{consume} : \mathsf{nat} \to o \\
\qquad \cdot \multimap B\ \textbf{as}\ Q.\ \begin{bmatrix} !\forall e\colon \mathsf{nat}.\ \ \mathit{produce}(e) \ \multimap B.\mathit{enq}(e) \\ \qquad\qquad\qquad ! \ \mathit{consume_req} \multimap \begin{bmatrix} B.\mathit{deq_req}, \\ \forall e\colon \mathsf{nat}.\ B.\mathit{deq}(e) \multimap \mathit{consume}(e) \end{bmatrix} \end{bmatrix} \\
\textbf{end}
\end{array}$$

The argument Q of prodcons is the name of a module with interface QUEUE. The following code uses queue defined earlier, and triggers two producers and one consumer by passing them the appropriate interface predicates exported by the module.

$$\cdot \multimap B\ \textbf{as}\ \mathsf{prodcons}(\mathsf{queue}).\ \begin{bmatrix} p_1(B.\mathit{produce}), p_2(B.\mathit{produce}), \\ c_1(B.\mathit{consume\_req}, B.\mathit{consume}) \end{bmatrix}$$

Modules in our language can do a lot more than taking other modules as arguments. For example, a networking module may itself make use of the functionalities of our producer-consumer module. We may therefore make the networking module parametric with respect to the specific implementation of the producer-consumer module. In ML parlance, this would be a functor that takes another functor as a parameter — a higher-order functor (something that is available in some extensions of Standard ML, but not in the official definition [19]). By the same token, nothing prevents us from defining a module that uses an instance of itself in some of its rules — a recursive module.

4 Multiset Rewriting with Modules

The module infrastructure we developed in the previous section had two parts:

1. We introduced convenience syntax for the programming patterns of $\mathcal{L}^{1.5}$ that realized module definitions ($\textbf{module}\ \ldots\ \textbf{end}$), module instantiations ($N\ \textbf{as}\ p\,t.\ \ldots$), and the use of exported names (e.g., $N.\mathit{add}$).
2. We imposed a series of restrictions on where and how predicates can be used (through the $\textbf{in}$ and $\textbf{out}$ markers), as well as a correspondence between the names exported by a module and the compound names used in client code.

We call the extension of $\mathcal{L}^{1.5}$ with both aspects $\mathcal{L}^{M}$. In this section, we describe how the added syntax (1) can be compiled away, thereby showing that $\mathcal{L}^{M}$ is just syntactic sugar for $\mathcal{L}^{1.5}$ — $\mathcal{L}^{1.5}$ has already all the ingredients to write modular code. We call this process *elaboration*. We handle the restrictions (2) by typechecking $\mathcal{L}^{M}$ code directly, as a user could violate them even if her code elaborates to a valid $\mathcal{L}^{1.5}$ program. In [5] we describe an extension of the typing infrastructure of $\mathcal{L}^{1.5}$ that checks that these restrictions as satisfied at the level of $\mathcal{L}^{M}$. Our module language is actually more flexible than what we saw in the examples of Section 3.

Once an $\mathcal{L}^{M}$ program has been typechecked, it is elaborated into an $\mathcal{L}^{1.5}$ program by compiling the module-specific constructs into native syntax. We then execute this $\mathcal{L}^{1.5}$ program.

The general syntax of a module definition has the form

$\mathbf{module}\ p\,(\Sigma_{par})$	Module name and parameters
$\quad\mathbf{provide}\ \ \Sigma_{export}$	Exported names
$\quad\mathbf{local}\ \ \Sigma_{local}$	Local predicates and constructors
$\quad P$	Module definition
$\mathbf{end}$	

The module interface is Σ_{export}. The modes of the exported predicate names (**in** and **out** in Section 3) are irrelevant after type checking — we ignore them. Let Σ^* denote the tuple of the names declared in signature Σ. Then this construct is elaborated into the $\mathcal{L}^{1.5}$ rule

$$\forall\Sigma_{par}.\,\forall\Sigma_{export}.\,p\,(\Sigma^*_{par},\Sigma^*_{export})\multimap\exists\Sigma_{local}.\,P$$

where the notation $\forall\Sigma.\,R$ prefixes the rule R with one universal quantification for each declaration in Σ, and similarly for $\exists\Sigma.\,P$.

Let Σ_{export} be the interface exported by the module p, defined as in the previous paragraph. For convenience, we express Σ_{export} as $\vec{X}:\vec{\tau}$, where the i-th declaration in Σ_{export} is $X_i:\tau_i$. Then, we elaborate the derived syntax for module instantiation,

$$N\ \mathbf{as}\ p\,t.\,P \qquad \text{into} \qquad \exists\Sigma_{export}.\,p(t,\vec{X}),[\vec{X}/N.\vec{X}]P$$

where $[\vec{X}/N.\vec{X}]P$ replaces each occurrence of $N.X_i$ in P with X_i. We implicitly assume that variables are renamed as to avoid capture.

Elaboration transforms a valid $\mathcal{L}^M$ program into a valid $\mathcal{L}^{1.5}$ program, ready to be executed. In particular, it removes all compound names of the form $N.X$.

5 Future Work and Conclusions

In this paper, we developed an advanced module system for $\mathcal{L}^1$, a small rule-based programming language. $\mathcal{L}^1$ corresponds to a large fragment of polarized intuitionistic linear logic under a derivation strategy based on focusing. Modules are rewrite rules of a specific form and can therefore be compiled away. Yet, they share many of the features of the module system of Standard ML, and provide several more. $\mathcal{L}^1$ incorporates key features of languages based on multiset rewriting [10,22], forward-chaining logic programming [12,14,2,7,8], and many process algebras [20,9]. Therefore, the module infrastructure developed in this paper can be applied directly to any of these formalisms.

Our immediate next step will be to implement our module system in the CoMingle system [15]. CoMingle is a rule-based framework aimed at simplifying the development of applications distributed over multiple Android devices. CoMingle programs are compiled, and one of the most interesting challenges of extending it with modules will be to support separate compilation, one of the key feature of advanced module systems à la Standard ML. We also intend to further study the languages defined in this paper. In particular, we want to investigate uses of second-order quantification to support reflection and runtime rule construction without sacrificing performance. We are also keen on adapting reasoning techniques commonly used in process calculi, for example bisimulation [20], to our language as $\mathcal{L}^1$ supports many of their features.

References

1. M. N. Abdallah. Procedures in Horn-clause Programming. In *ICLP'86*, pages 433–447. Springer LNCS 225, 1986.
2. M. Ashley-Rollman, P. Lee, S. Goldstein, P. Pillai, and J. Campbell. A Language for Large Ensembles of Independently Executing Nodes. In *ICLP'09*. Springer LNCS 5649, 2009.
3. J.-P. Banâtre and D. Le Métayer. Programming by Multiset Transformation. *Communications of the ACM*, 36(1):98–111, 1993.
4. M. Bugliesi, E. Lamma, and P. Mello. Modularity in Logic Programming. *The Journal of Logic Programming*, 19–20(1):443–502, 1994.
5. I. Cervesato and E. S. Lam. Modular Multiset Rewriting in Focused Linear Logic. Technical Report CMU-CS-15-117, Carnegie Mellon University, Pittsburgh, PA, 2015.
6. I. Cervesato and A. Scedrov. Relating State-Based and Process-Based Concurrency through Linear Logic. *Information & Computation*, 207(10):1044–1077, 2009.
7. F. Cruz, M. Ashley-Rollman, S. Goldstein, R. Rocha, and F. Pfenning. Bottom-Up Logic Programming for Multicores. In *DAMP'12*, 2012.
8. F. Cruz, R. Rocha, S. Goldstein, and F. Pfenning. A Linear Logic Programming Language for Concurrent Programming over Graph Structures. In *ICLP'14*, Vienna, Austria, 2014.
9. C. Fournet. *The Join-Calculus: a Calculus for Distributed Mobile Programming*. PhD thesis, École Polytechnique, Palaiseau, 1998.
10. T. Frühwirth. *Constraint Handling Rules*. Cambridge University Press, 2009.
11. M. J. Gabbay and A. Mathijssen. One-and-a-halfth-order Logic. *Journal of Logic and Computation*, 18(4):521–562, August 2008.
12. H. Gallaire, J. Minker, and J. M. Nicolas. Logic and Databases: A Deductive Approach. *ACM Computing Survey*, 16(2):153–185, June 1984.
13. J.-Y. Girard. Linear Logic. *Theoretical Computer Science*, 50:1–102, 1987.
14. S. Grumbach and F. Wang. Netlog, a Rule-based Language for Distributed Programming. In *PADL'10*, pages 88–103. Springer LNCS 5937, 2010.
15. E. S. Lam, I. Cervesato, and N. F. Haque. Comingle: Distributed Logic Programming for Decentralized Mobile Ensembles. In *COORDINATION'15*. Springer LNCS 9037, 2015.
16. O. Laurent, M. Quatrini, and L. Tortora de Falco. Polarized and Focalized Linear and Classical Proofs. *Annals of Pure and Applied Logic*, 134(2–3):217–264, July 2005.
17. J. Meseguer and C. Braga. Modular Rewriting Semantics of Programming Languages. In *AMAST'04*, pages 364–378. Springer LNCS 3116, 2004.
18. D. Miller. A Proposal for Modules in λProlog. In *ELP'94*, pages 206–221. Springer LNCS 798, 1994.
19. R. Milner, R. Harper, D. MacQueen, and M. Tofte. *The Definition of Standard ML – Revised*. MIT Press, 1997.
20. D. Sangiorgi and D. Walker. *The Pi-Calculus — a Theory of Mobile Processes*. Cambridge University Press, 2001.
21. R. J. Simmons. Structural Focalization. *ACM Transaction in Computational Logic*, 15(3):1–33, Sept. 2014.
22. K. Watkins, F. Pfenning, D. Walker, and I. Cervesato. Specifying Properties of Concurrent Computations in CLF. In *LFM'04*, pages 67–87, Cork, Ireland, 2007. ENTCS 199.

Acknowledgments This paper was made possible by grants NPRP 09-667-1-100 (*Effective Programming for Large Distributed Ensembles*) and NPRP 4-341-1-059 (*Usable automated data inference for end-users*) from the Qatar National Research Fund (a member of the Qatar Foundation). The statements made herein are solely the responsibility of the authors.