Modular Multiset Rewriting in Focused Linear Logic

Iliano Cervesato and Edmund S. L. Lam

July 2015
CMU-CS-15-117
CMU-CS-QTR-128

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Carnegie Mellon University, Qatar campus.
The author can be reached at iliano@cmu.edu or sllam@qatar.cmu.edu .

Abstract

Traditional multiset rewriting arises from a fragment of polarized linear logic, deriving its semantics from focused proof search. In recent years, languages based on multiset rewriting have been used for ever more ambitious applications. As program size grows however, so does the overhead of team-based development and the need for reusing components. Furthermore, there is a point where keeping a large flat collection of rules becomes impractical. In this report, we propose a module system for a small logically-motivated rule-based language that subsumes traditional multiset rewriting. The resulting modules are nothing more than rewrite rules of a specific form, and therefore are themselves just formulas in logic. 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, parametrization by values and by other modules (functors in ML). Additionally, our modules offer features that are essential for concurrent programming, for example facilities of sharing private names. This work establishes a foundation for modularity in rule-based languages and is directly applicable to many languages based on multiset rewriting, most forward-chaining logic programming languages and many process algebras.

* This report was made possible by grant NPRP 09-667-1-100, *Effective Programming for Large Distributed Ensembles*, from the Qatar National Research Fund (a member of the Qatar Foundation). The statements made herein are solely the responsibility of the authors.

Keywords: Multiset Rewriting, Logic Programming, Modularity, Focused Search.

Contents

List of Figures

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. Indeed, special-purpose languages for security [23], networking [18, 24], robot coordination [2], multicore programming [11], graph programming [12] and mobile applications [21] have all recently been proposed as extensions of Datalog [16, 5], itself a rule-based database language founded on forward-chaining proof search. A few general-purpose languages based on this paradigm have also been proposed, for example the multiset rewriting language GAMMA [3] and more recently CHR [14]. The asynchronous state transformation model embodied by this paradigm has also been shown to subsume various models of concurrency [9], in particular multiset rewriting, Petri nets [30], process algebra [34], as well as the rare hybrid models [36, 8, 13].

As languages gain popularity, the need for modularity increases, since the overhead associated with writing code grows with program size. Yet, none of the languages just mentioned features a module system, even though some of them, for example CHR [14] and CoMingle [21], are starting to be used to write sizable applications. Modularity tames complexity. In traditional programming languages, it addresses the challenges of breaking a large program into a hierarchy of components with clear interfaces, implementation independence, team-based development, dependency management, code reuse, and separate compilation. These features alone justify extending popular rule-based languages with a module system.

Programming-in-the-large in a rule-based programming languages brings about additional challenges not typically found in imperative or functional languages. First, languages such as Datalog [16, 5], CHR [14] and GAMMA [3] have (like Prolog) a flat name space which gives no protections against accidentally reusing a name. Moreover, each rule in such a language adds to the definition of the names it contains rather than overriding them (as C does, for example). Second, these languages (like both C and Prolog) tend to have an open scope, meaning that there is no support for local definitions or private names. Finally, a rule in these languages 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 a moderate size difficult [7]. 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 report, we develop a module system for a small rule-based programming language. This language, which we call $\mathcal{L}^1$, is a fragment of the formalism in [9] and subsumes many languages founded on multiset rewriting, forward-chaining proof search and process algebra. 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 [35]. We extend $\mathcal{L}^1$ with a mild form of second-order quantification [15] to make our presentation more succinct (without adding expressiveness). We call this extension $\mathcal{L}^{1.5}$.

We engineer a module system for $\mathcal{L}^{1.5}$ 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 $\mathcal{L}^{1.5}$ we call $\mathcal{L}^M$. The module system of $\mathcal{L}^M$ support many of the facilities for modular programming found in Standard ML [25, 29], 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 [29]. Yet, because the modular constructs of $\mathcal{L}^M$ are just programming patterns in $\mathcal{L}^{1.5}$, programs in $\mathcal{L}^M$ can be faithfully compiled into $\mathcal{L}^{1.5}$ and thus 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 [26], 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 [27, 28] 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 [15].

The main contributions of this report are thus threefold:

1. 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 proposed in [9].
2. We engineer a powerful module system on top of this core language with support for name space separation, parametricity, and controlled name sharing.
3. We show that this module infrastructure is little more than syntactic sugar over the core language $\mathcal{L}^1$, and can therefore be compiled away.

It is worth repeating that, because a large number of rule-based languages, including forward-chaining logic programming languages such as Datalog and its offshoots, multiset rewriting, as well as many process algebras can be viewed as linguistic fragments of our core language [9], this work defines a general mechanism for augmenting any of them with a powerful yet declarative module system. In fact, this work provides a logical foundation of modularity in rule-based languages in general.

The remainder of this report is organized as follows: in Section 2 we define the core multiset rewriting language on which we will build our module system, we relate it to more traditional presentations of multiset rewriting, and we discuss its logical underpinning. In Section 3, we introduce our modular infrastructure through a series of examples, incrementally developing syntax for it. In Section 4, we collect the language just developed, discuss additional syntactic checks, and explain how to compile it to the core language. We explore related work in Section 5 and outlines future developments in Section 6. Appendix A summarizes the syntax and semantics of our language and Appendix B lays down the details of its logical underpinning.

2 Core Language

This section develops a small, logically-motivated, rule-based language that will be used throughout the report. This formalism, a conservative extension of traditional multiset rewriting, will act as the core language in which we write (non-modular) programs. It is also the language our modular infrastructure will compile into. Section 2.1 defines this language, Section 2.2 highlights its foundations in linear logic, and Section 2.3 enriches it with a mild extension that will simplify writing programs in Section 3. As our development is incremental, we collect and organize the definition of the overall language in Appendix A.

2.1 Multiset Rewriting with Existentials and Nested Rules

Our core formalism 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 [9]. Because we are interested in writing actual programs, we consider a simply-typed variant. We will refer to the language discussed in this section as $\mathcal{L}^1$.

We now introduce the syntax of $\mathcal{L}^1$ in Section 2.1.1, give typing rules for it in Section 2.1.2, present its operational semantics in Section 2.1.3, and situate it with respect to other formalisms in Section 2.1.4.

2.1.1 Syntax

The syntax of the language $\mathcal{L}^1$ is specified by the following grammar:

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

Our formalism is layered into a language of terms, used for describing entities of interest, and a language of formulas, used to specify computations.

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 — they correspond to the constants of a first-order term language. 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 atoms, denoted o. The type constructors, products and function types, match the term constructors, with the type of $()$ written $\top$. While this minimal typing infrastructure is sufficient for the formal development in this report, it can be considerably enriched with additional type and term constructors — in fact, many of the examples in Section 4 will make use of the type nat of the unary natural numbers and the associated constructors.

The language of formulas consists of programs, themselves built out of rules and left-hand sides and ultimately atoms. An *atom* A is a predicate symbol p applied to a term. As with function symbols, we abbreviate $p\,()$ in examples as just p — this is a propositional atom.

Atoms are used to build *rules*, R, which are the universal closures of rewrite directives 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. Considering the operator "," commutative and associative with unit "$\cdot$" will simplify our presentation, although this is not strictly necessary from an operational perspective. The right-hand side P of a rewrite directive is a multiset of either atoms A, reusable 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 single-use and reusable atoms and rules, and of existentially quantified programs.

The quantifiers $\forall x : \tau. R$ and $\exists x : \tau. P$ are the only binders in $\mathcal{L}^1$ (although richer languages could have more). We adopt the standard definition of free and bound variables (the above operators bind the variable x with scope R and P, respectively). A term or atom is *ground* if it does not contain free variables. A rule or program is *closed* if it does not contain free variables. Bound variables can be renamed freely as long as they do not induce capture. Given a syntactic entity O possibly containing a free variable x, we write $[t/x]O$ for the capture-avoiding substitution of term t for every free occurrence of x in O. As we perform a substitution, we will always be careful that the variable x and the term t have the same type (which for the time being can only be ι). 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.

From the above grammar, it is apparent that traditional first-order multiset rewriting is the fragment of $\mathcal{L}^1$ where the right-hand side of a rule is a multiset of atoms (thereby disallowing rules and existentials) and where a program is a multiset of reusable rules. Maybe more surprisingly, this formalism also subsumes many process algebras: for example the asynchronous π-calculus corresponds to the restriction where a left-hand side always consists of exactly one atom (corresponding to a receive action) and where "," is interpreted as parallel composition. This correspondence is explored further in Section 2.1.4 and at much greater depth in [9].

2.1.2 Typing

Function and predicate symbols have types $\tau \to \iota$ and $\tau \to o$ respectively. The symbols in use during execution together with their type are collected into a *signature*, denoted Σ. A signature induces a notion of type validity for the various entities of our language, which is captured by the typing judgments and rules in this section.

As we traverse universal quantifier, the typing rules collect the corresponding variables and their types into a *context*, denoted Γ. (Existential variables are treated as fresh function symbols and recorded in the signature during type checking.) Signatures and contexts are defined as follows:

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

Notice that context variables have type ι and therefore stand for terms.

The typing semantics of $\mathcal{L}^1$ is specified by the following judgments:

Terms	$\Gamma \vdash_\Sigma t : \tau$	*Term t has type τ in Γ and Σ*
Atoms	$\Gamma \vdash_\Sigma A\ atom$	*A is a well-typed atom in Γ and Σ*
Left-hand sides	$\Gamma \vdash_\Sigma l\ lhs$	*l is a well-typed left-hand side in Γ and Σ*
Rules	$\Gamma \vdash_\Sigma R\ rule$	*R is a well-typed rule in Γ and Σ*
Programs	$\Gamma \vdash_\Sigma P\ prog$	*P is a well-typed program in Γ and Σ*

The fairly conventional rules defining these judgments are given in Figure 2.1.

Valid rewrite directives $l \multimap P$ are subject to the requirement that the free variables in P shall occur in l or in the left-hand side of an enclosing rule. While this *safety requirement* is not enforced by the rules in Figure 2.1, Appendix A.3 defines and extension that does enforce safety.

2.1.3 Operational Semantics

Computation in $\mathcal{L}^1$ takes the form of state transitions. A *state* is a triple $\Sigma.\langle \Omega \ ; \ \Pi \rangle$ consisting of a collection Ω of ground atoms and closed rules we call the *archive*, of a closed program Π, and a signature Σ that accounts for all the function and predicate symbols in Ω and Π. We emphasize that the program must be closed by writing it as Π rather

Terms: $\dfrac{}{\Gamma, x : \tau \vdash_\Sigma x : \tau} \qquad \dfrac{\Gamma \vdash_{\Sigma, f:\tau\to\iota} t : \tau}{\Gamma \vdash_{\Sigma, f:\tau\to\iota} f\ t : \iota} \qquad \dfrac{\Gamma \vdash_\Sigma t_1 : \tau_1 \quad \Gamma \vdash_\Sigma t_2 : \tau_2}{\Gamma \vdash_\Sigma (t_1, t_2) : \tau_1 \times \tau_2} \qquad \dfrac{}{\Gamma \vdash_\Sigma () : \top}$

Atoms: $\dfrac{\Gamma \vdash_{\Sigma, p:\tau\to o} t : \tau}{\Gamma \vdash_{\Sigma, p:\tau\to o} p\ t\ \mathit{atom}}$

LHS: $\dfrac{}{\Gamma \vdash_\Sigma \cdot\ \mathit{lhs}} \qquad \dfrac{\Gamma \vdash_\Sigma A\ \mathit{atom} \quad \Gamma \vdash_\Sigma l\ \mathit{lhs}}{\Gamma \vdash_\Sigma A, l\ \mathit{lhs}}$

Rules: $\dfrac{\Gamma \vdash_\Sigma l\ \mathit{lhs} \quad \Gamma \vdash_\Sigma P\ \mathit{prog}}{\Gamma \vdash_\Sigma l \multimap P\ \mathit{rule}} \qquad \dfrac{\Gamma, x : \iota \vdash_\Sigma R\ \mathit{rule}}{\Gamma \vdash_\Sigma \forall x : \iota.\, R\ \mathit{rule}}$

Programs: $\dfrac{}{\Gamma \vdash_\Sigma \cdot\ \mathit{prog}} \qquad \dfrac{\Gamma \vdash_\Sigma P_1\ \mathit{prog} \quad \Gamma \vdash_\Sigma P_2\ \mathit{prog}}{\Gamma \vdash_\Sigma P_1, P_2\ \mathit{prog}}$

$$\frac{\Gamma \vdash_\Sigma A\ \mathit{atom}}{\Gamma \vdash_\Sigma A\ \mathit{prog}} \qquad \frac{\Gamma \vdash_\Sigma A\ \mathit{atom}}{\Gamma \vdash_\Sigma !A\ \mathit{prog}} \qquad \frac{\Gamma \vdash_\Sigma R\ \mathit{rule}}{\Gamma \vdash_\Sigma R\ \mathit{prog}} \qquad \frac{\Gamma \vdash_\Sigma R\ \mathit{rule}}{\Gamma \vdash_\Sigma !R\ \mathit{prog}}$$

$$\frac{\Gamma \vdash_{\Sigma, x:\tau\to\iota} P\ \mathit{prog}}{\Gamma \vdash_\Sigma \exists x : \tau \to \iota.\, P\ \mathit{prog}}$$

Figure 2.1: Typing Rules for $\mathcal{L}^1$

than P. Archives will be used to store reusable components of the program Π as the computation unfolds in Π itself. We define archives next, and copy the definition of (closed) programs using this notation for emphasis:

$$\begin{array}{rrcl} \textit{Archives} & \Omega & ::= & \cdot \mid \Omega, A \mid \Omega, R \\ \textit{State programs} & \Pi & ::= & \cdot \mid \Pi, \Pi \mid A \mid !A \mid R \mid !R \mid \exists x : \tau \to \iota.\, P \\ \textit{State} & \Sigma.\langle \Omega\ ;\ \Pi \rangle & & \end{array}$$

We will occasionally refer to Π itself as a state when the other components are of secondary concern. When talking about states, it will be important that "," be commutative and associative with unit "$\cdot$". This induces a notion of equivalence among states — a congruence indeed when applied to embedded programs. This congruence is examined in Appendix A.6. Note that all atoms A appearing in both the archive and the program component of a state must be ground. Furthermore, since rules R are closed and have the form $\forall \vec{x} : \vec{\tau}.\, (l \multimap P)$ we will occasionally use the notation $\forall (l \multimap P)$ with the understanding that the universal quantification is over all the free variables in $l \multimap P$.

The above definition combines elements of the notion of "state" found in both multiset rewriting and process algebras. In multiset rewriting, a state is traditionally understood a multiset of ground atoms that carry data (in the form of terms) on which a separate notion of immutable rules perform computation through rewriting. In process algebra, the "state" is a collection of processes that interact with each other as the computation takes place. In $\mathcal{L}^1$, we have both: the ground atoms in Ω and Π act as data while non-atomic state items in them give rise to computation, like rules in multiset rewriting or processes in process algebra.

We extend the typing infrastructure discussed in the last section with the judgments

Archives	$\vdash_\Sigma \Omega\ \mathit{archive}$	*Archive Ω is well typed in Σ*
States	$\vdash \Sigma.\langle \Omega\ ;\ \Pi \rangle\ \mathit{state}$	*State $\Sigma.\langle \Omega\ ;\ \Pi \rangle$ is well typed*

to describe typing over archives and valid states. A state $\Sigma.\langle \Omega \ ; \ \Pi \rangle$ is valid if both the archive Ω and the program Π are well typed in the empty context and Σ, as captured by the following rules:

$$\frac{}{\vdash_\Sigma \cdot \ \mathit{archive}} \qquad \frac{\vdash_\Sigma \Omega \ \mathit{archive} \quad \cdot \vdash_\Sigma A \ \mathit{atom}}{\vdash_\Sigma \Omega, A \ \mathit{archive}} \qquad \frac{\vdash_\Sigma \Omega \ \mathit{archive} \quad \cdot \vdash_\Sigma R \ \mathit{rule}}{\vdash_\Sigma \Omega, R \ \mathit{archive}}$$

$$\frac{\vdash_\Sigma \Omega \ \mathit{archive} \quad \cdot \vdash_\Sigma \Pi \ \mathit{prog}}{\vdash \Sigma.\langle \Omega \ ; \ \Pi \rangle \ \mathit{state}}$$

We will give two characterizations of the rewriting semantics of $\mathcal{L}^1$. The first interprets each operator in $\mathcal{L}^1$ as an independent state transformation directive. This makes it very close to traditional presentations of linear logic (see Section 2.2) but the resulting semantics is overly non-deterministic. The second presentation is instead focused on applying rules fully, and does so by interpreting them as meta-transformations, with the effect of giving a strong operational flavor to the language and an effective strategy for implementing an interpreter for it. This second characterization accounts for the standard semantics found in many languages based on multiset rewriting, forward chaining and process transformation. It also has its roots in linear logic, but this time in its focused presentation [22]. These two characterizations are not equivalent as the latter leads to fewer reachable states. It is however preferable as these states match the intuition of what is traditionally meant by applying a rewrite rule, leaving out states littered with half-applied rules. We will examine how this relates to the corresponding presentations of linear logic in Section 2.2.

Unfocused Rewriting Semantics The unfocused evaluation semantics makes use of a step judgment of the form

$$\Sigma.\langle \Omega \ ; \ \Pi \rangle \mapsto \Sigma'.\langle \Omega' \ ; \ \Pi' \rangle \qquad \textit{State } \Sigma.\langle \Omega \ ; \ \Pi \rangle \textit{ transitions to state } \Sigma'.\langle \Omega \ ; \ \Pi' \rangle \textit{ in one step}$$

In this semantics, each operator in the language is understood as a directive to carry out one step of computation in a given state. Therefore, each operator yields one transition rule, given in the following table:

$$\begin{array}{rcll}
\Sigma.\langle \Omega, l_1 \ ; \ \Pi, l_2, (l_1, l_2) \multimap P \rangle & \mapsto & \Sigma.\langle \Omega, l_1 \ ; \ \Pi, P \rangle & \\
\Sigma.\langle \Omega \ ; \ \Pi, \forall x : \iota.\, R \rangle & \mapsto & \Sigma.\langle \Omega \ ; \ \Pi, [t/x]R \rangle & \text{if } \cdot \vdash_\Sigma t : \iota \\
\Sigma.\langle \Omega \ ; \ \Pi, !A \rangle & \mapsto & \Sigma.\langle \Omega, A \ ; \ \Pi \rangle & \\
\Sigma.\langle \Omega \ ; \ \Pi, !R \rangle & \mapsto & \Sigma.\langle \Omega, R \ ; \ \Pi \rangle & \\
\Sigma.\langle \Omega \ ; \ \Pi, \exists x : \tau \to \iota.\, P \rangle & \mapsto & (\Sigma, x : \tau \to \iota).\langle \Omega \ ; \ \Pi, P \rangle & \\
\Sigma.\langle \Omega, R \ ; \ \Pi \rangle & \mapsto & \Sigma.\langle \Omega, R \ ; \ \Pi, R \rangle &
\end{array}$$

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. It retrieves reusable atoms from the archive and single-use atoms from the program component of the state. Notice that the atom in the program side are consumed but atoms in the archive side are retained and can therefore be used over and over. The operator $\forall$ is an instantiation directive: it picks a term of the appropriate type and replaces the bound variable with it. Instead, ! is to all effects a replication directive: it installs the entity it prefixes in the archive, enabling repeated accesses to atoms by the rewrite directive and to rules in the last transition, which copies a rule to the program side while retaining the master copy in the archive. Finally, $\exists$ is a name generation directive which installs a new symbol of the appropriate type in the state's signature.

Evaluation steps transform valid states into valid states, a property captured by the following lemma. In particular, it maintains the invariant that a state is closed.

Lemma 1 (Type preservation) *If* $\vdash \Sigma.\langle \Omega \ ; \ \Pi \rangle \ \mathit{state}$ *and* $\Sigma.\langle \Omega \ ; \ \Pi \rangle \mapsto \Sigma'.\langle \Omega' \ ; \ \Pi' \rangle$*, then* $\vdash \Sigma'.\langle \Omega' \ ; \ \Pi' \rangle \ \mathit{state}$.

Proof The proof proceeds by cases on the step rule applied and then by inversion on the typing derivation. It relies on state equivalence and the corresponding equivalence it induces over typing derivation. □

We write $\Sigma.\langle \Omega \ ; \ \Pi \rangle \mapsto^* \Sigma'.\langle \Omega' \ ; \ \Pi' \rangle$ for the reflexive and transitive closure of the above judgment, meaning that state $\Sigma.\langle \Omega \ ; \ \Pi \rangle$ transitions to $\Sigma'.\langle \Omega' \ ; \ \Pi' \rangle$ in zero or more steps. Naturally, type preservation holds for this iterated form as well.

The rules of this semantics are pleasantly simple as they tease out the specific behavior of each individual language construct. They are also highly non-deterministic as any of them is applicable whenever the corresponding operator appears as the top-level constructor of a state element. These transitions stem from the logic interpretation of our operators, and can in fact be read off directly from the rules of linear logic [9] — see Appendix B.3. This further supports the orthogonality of our constructs.

Focused Rewriting Semantics By considering each operator in isolation when applying a step, the above semantics falls short of the expected behavior of a rewriting language. For example, consider the state

$$(\underbrace{p : \iota \to o, q : \iota \to o, a : \iota, b : \iota}_{\Sigma}).\langle \cdot \; ; \; \underbrace{p\, a, \forall x : \iota.\, p\, x \multimap q\, x}_{\Pi} \rangle$$

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 and the empty archive (which do not change) for clarity. 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 operational 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 have the potential to block other currently available transformations, while others do not. In particular, turning a top-level existential variable into a new name or moving a reusable entity to the archive never preempt other available transitions.

A closed program without top-level existential quantifiers or reusable items is called *stable*. We write Π for a state program Π that is stable. A stable program is therefore defined as follows:

Stable state program	Π	$::=$	$\cdot$ empty state
		$\mid$	Π, A atom
		$\mid$	Π, R rule

A *stable state* has the form $\Sigma.\langle \Omega \; ; \; \Pi \rangle$.

The focused operational semantics is expressed by the following two judgments:

$\Sigma.\langle \Omega \; ; \; \Pi \rangle \Mapsto \Sigma'.\langle \Omega' \; ; \; \Pi' \rangle$ *Non-stable state* $\Sigma.\langle \Omega \; ; \; \Pi \rangle$ *transitions to state* $\Sigma'.\langle \Omega' \; ; \; \Pi' \rangle$ *in one step*

$\Sigma.\langle \Omega \; ; \; \Pi \rangle \Mapsto \Sigma'.\langle \Omega' \; ; \; \Pi' \rangle$ *Stable state* $\Sigma.\langle \Omega \; ; \; \Pi \rangle$ *transitions to state* $\Sigma'.\langle \Omega' \; ; \; \Pi' \rangle$ *in one step*

The first judgment is realized by selecting an existential or reusable program component in Π and processing it by eliminating the quantifier and creating a new symbol and by moving its immediate subformula to the archive, respectively. By iterating this step, we will reach a stable state in finite time. At this point, the second judgment is applied. It selects a single-use or a reusable rule from the program or archive part of the state 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. Observe that some left-hand side atoms are retrieved from the archive and others from the program component of the state. The resulting state replaces it with the corresponding instance $P\theta$ of the right-hand side P. This state may not be stable as $P\theta$ could contain existentially quantified and reusable components. The following transitions formalize this insight.

$$\begin{aligned}
\Sigma.\langle \Omega \; ; \; \Pi, !A \rangle &\Mapsto \Sigma.\langle \Omega, A \; ; \; \Pi \rangle \\
\Sigma.\langle \Omega \; ; \; \Pi, !R \rangle &\Mapsto \Sigma.\langle \Omega, R \; ; \; \Pi \rangle \\
\Sigma.\langle \Omega \; ; \; \Pi, \exists x : \tau \to \iota.\, P \rangle &\Mapsto (\Sigma, x : \tau \to \iota).\langle \Omega \; ; \; \Pi, P \rangle \\
\Sigma.\langle \Omega, l_1\theta \; ; \; \Pi, l_2\theta, \forall(l_1, l_2 \multimap P) \rangle &\Mapsto \Sigma.\langle \Omega, l_1\theta \; ; \; \Pi, P\theta \rangle \\
\Sigma.\langle \underbrace{\Omega, l_1\theta, \forall(l_1, l_2 \multimap P)}_{\Omega^*} \; ; \; \Pi, l_2\theta \rangle &\Mapsto \Sigma.\langle \Omega^* \; ; \; \Pi, P\theta \rangle
\end{aligned}$$

As usual, we write $\mapsto^*$ and $\Mapsto^*$ for the reflexive and transitive closure of these relations.

Type preservation holds for the base relations and their transitive closure. 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.

Lemma 2

1. *If* $\Sigma.\langle \Omega \ ; \ \Pi \rangle \Mapsto \Sigma'.\langle \Omega' \ ; \ \Pi' \rangle$, *then* $\Sigma.\langle \Omega \ ; \ \Pi \rangle \mapsto \Sigma'.\langle \Omega' \ ; \ \Pi' \rangle$.
2. *If* $\Sigma.\langle \Omega \ ; \ \Pi \rangle \Mapsto \Sigma'.\langle \Omega' \ ; \ \Pi' \rangle$, *then* $\Sigma.\langle \Omega \ ; \ \Pi \rangle \mapsto^* \Sigma'.\langle \Omega' \ ; \ \Pi' \rangle$.

Proof The first part simply uses the isomorphic rule of the unfocused semantics. The second proceeds by induction on the structure of the selected rule. □

The reverse does not hold, as we saw earlier.

Because rules R are applicable only in stable states, it is typical to follow up such an application with zero or more transitions of the first kind to reach the next stable state. We write $\Rrightarrow$ for this relation: $\Sigma.\langle \Omega \ ; \ \Pi \rangle \Rrightarrow \Sigma'.\langle \Omega' \ ; \ \Pi' \rangle$ is defined as $\Sigma.\langle \Omega \ ; \ \Pi \rangle \Mapsto \Sigma.\langle \Omega_1 \ ; \ \Pi_1 \rangle \Mapsto^* \Sigma'.\langle \Omega' \ ; \ \Pi' \rangle$.

2.1.4 Relationship to Other Languages

The language $\mathcal{L}^1$ is a syntactic fragment of the formalism of ω-multisets examined in [9] as a logical reconstruction of multiset rewriting and some forms of process algebra. It instantiates ω-multisets by fixing a language of terms as discussed earlier, while [9] kept it open-ended (requiring only predicativity). 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 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 forward logic programming languages we are aware of, and is much more directly supported by a focusing view of proof search (see Section 2.2). Therefore, $\mathcal{L}^1$ occupies a sweet spot between the freewheeling generality of ω-multiset rewriting and the implementation simplicity of many rule-based languages. Differently from ω-multisets, which are freely generated from the available connectives, $\mathcal{L}^1$ limits the usage of the ! operator to just rules and right-hand side atoms. These restrictions avoid expressions that are of little use in programming practice (for example doubly reusable rules $!!R$ or left-hand side atoms of the form $!A$). We also left out the choice operator of ω-multisets (written $\&$) because we did not need it in any of the examples in this report. Adding it back is straightforward.

The language $\mathcal{L}^1$ subsumes various rule-based formalisms founded on multiset rewriting, forward proof search and process algebra as a syntactic fragment. First-order multiset rewriting, as found for example in CHR [14], relies on rules whose right-hand side is a multiset of atoms and that can be used arbitrarily many times, and therefore correspond to $\mathcal{L}^1$ rules of the form $!\forall \vec{x}.\, (l_1 \multimap l_2)$. Languages such as MSR [6] 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 $!\forall \vec{x}.\, (l_1 \multimap \exists \vec{y}.\, l_2)$. Datalog clauses [16, 5] are written in $\mathcal{L}^1$ as $!\forall \vec{x}.\, (l \multimap !A)$ while their facts are reusable program atoms as this language monotonically grows the set of known facts during computation.

As shown in [9], 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 multiset union is commutative and associative with unit the empty multiset (thereby accounting for the analogous properties of parallel composition and the inert process). The asynchronous π-calculus [34] 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). Its synchronous variant can be simulated in $\mathcal{L}^1$ through a shallow encoding [8].

Our language, like ω-multiset rewriting itself, contains fragments that correspond to both the state transition approach to specifying concurrent computations (as multiset rewriting and Petri nets for example) and specifications in the process-algebraic style. It in fact supports hybrid specifications as well, as found in the Join calculus [13] and in CLF [8, 36].

2.2 Logical Foundations

The language $\mathcal{L}^1$, like ω-multisets [9], corresponds exactly to a fragment of intuitionistic linear logic [17]. 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. In this section, we outline this correspondence, with a much more detailed discussion relegated to Appendix B.

The operators "," and "·", $\multimap$, !, $\forall$ 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). Because the left-hand side of a rule in $\mathcal{L}^1$ consists solely of atoms and only rules and atoms are reusable, the actual MEILL formulas our language maps to are similarly restricted — the exact correspondence is found in Appendix B.3. By comparison, ω-multiset rewriting also includes additive conjunction, written $\&$, and allows freely combining these operators, which matches exactly the corresponding logic.

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, for an appropriate presentation of the exponential !. 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 in $\mathcal{L}^1$ — the rules for this judgment are given in Appendix B.1. Consider for example the transition for an existential $\mathcal{L}^1$ program and the left sequent rule $\exists\mathsf{L}$ for the existential quantifier:

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

The antecedent $\Omega \;;\; \Pi, \exists x : \tau \to \iota.\, P$ of the transition corresponds to the contexts $\Gamma; \Delta, \exists x : \tau \to \iota.\, \varphi$ of the rule conclusion, while its consequent $\Omega \;;\; (\Pi, P)$ matches the contexts of the premise $\Gamma; (\Delta, \varphi)$ 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. The interplay between persistent formulas $!\varphi$ and the persistent context Γ matches the rules for reusable entities in $\mathcal{L}^1$ [9]. Altogether, this correspondence is captured by the following property, proved in [9] and discussed further in Appendix B.3, where $\lceil\Omega\rceil$ and $\lceil\Pi\rceil$ are MEILL contexts corresponding to archive Ω and state program Π, and the formula $\exists\Sigma'.\ulcorner!\Omega'; \Pi'\urcorner$ is obtained by taking the multiplicative conjunction of the encodings of the entities in Π' and in Ω' (the latter are embedded within a !) and then prefixing the result with an existential quantification for each declaration in the signature Σ'.

Theorem 3 (Soundness) *If* $\vdash \Sigma.\langle \Omega \;;\; \Pi\rangle$ *state and* $\Sigma.\langle \Omega \;;\; \Pi\rangle \mapsto \Sigma'.\langle \Omega' \;;\; \Pi'\rangle$, *then* $\lceil\Omega\rceil; \lceil\Pi\rceil \longrightarrow_\Sigma \exists\Sigma'.\ulcorner!\Omega'; \Pi'\urcorner$.

This correspondence between the left rules of linear logic and rewriting directives was observed in [9] for the larger formalism of ω-multiset rewriting.

The transitions of the focused rewriting semantics of $\mathcal{L}^1$ originate from the focused presentation of linear logic [22], 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 [35] and for linear logic [22], and specifically for MEILL — see Appendix B.2 for a detailed definition. Transitions of the form $\Sigma.\langle \Omega \;;\; \Pi\rangle \Rrightarrow \Sigma'.\langle \Omega' \;;\; \Pi'\rangle$ correspond to invertible rules and are handled as in the unfocused case. Instead, transitions of the form $\Sigma.\langle \Omega \;;\; \Pi\rangle \Rrightarrow \Sigma'.\langle \Omega' \;;\; \Pi'\rangle$ correspond to the selection of a focus formula and the consequent chaining phase. Consider for example the transition for a single-use rule $\forall(l_1, l_2 \multimap P)$. The derivation snippet below selects a context formula $\forall(\varphi_{l_1} \otimes \varphi_{l_2} \multimap \varphi_P)$ of the corresponding form from a sequent where no invertible rule is applicable (generically written $\Gamma; \Delta \Longrightarrow_\Sigma \psi$), 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 formulas $\varphi_{l_i}\theta$ are conjunctions of atomic formulas matching $l_i\theta$, which allows continuing the chaining phase in the left premise of rule $\multimap\mathsf{L}$ into a complete derivation when Γ_1 and Δ_2 consist exactly of the atoms in l_1 and l_2, respectively. The translation φ_P of the program P re-enables an invertible rule and therefore the derivation loses focus in the rightmost open

premise.

$$\Sigma.\langle \Omega, l_1\theta \;;\; \Pi, l_2\theta, \forall(l_1, l_2 \multimap P)\rangle \;\Mapsto\; \Sigma.\langle \Omega, l_1\theta \;;\; \Pi, P\theta\rangle \quad \rightsquigarrow$$

$$\dfrac{\dfrac{\cdots \quad \overline{\Gamma, \Gamma_1\theta; A\theta \Longrightarrow_\Sigma \boxed{A\theta}}^{\mathsf{atmL}} \quad \cdots \quad \overline{\Gamma, \Delta_1'\theta, A\theta; \cdot \Longrightarrow_\Sigma \boxed{A\theta}}^{\mathsf{atm!L}} \quad \cdots \quad \dfrac{\Gamma, \Gamma_1\theta; \Delta, P\theta \Longrightarrow_\Sigma \psi}{\Gamma, \Gamma_1\theta; \Delta, \boxed{P\theta} \Longrightarrow_\Sigma \psi}\,\mathsf{blurL}}{\Gamma, \Gamma_1\theta; \Delta_2\theta \Longrightarrow_\Sigma \boxed{\varphi_{l_1}\theta \otimes \varphi_{l_2}\theta}} \quad \vdots}{\Gamma, \Gamma_1\theta; \Delta, \Delta_2\theta, \boxed{(\varphi_{l_1} \otimes \varphi_{l_2} \multimap \varphi_P)\theta} \Longrightarrow_\Sigma \psi}\,\multimap\mathsf{L}$$

$$\dfrac{\Gamma, \Gamma_1\theta; \Delta, \Delta_2\theta, \boxed{(\varphi_{l_1} \otimes \varphi_{l_2} \multimap \varphi_P)\theta} \Longrightarrow_\Sigma \psi}{\Gamma, \Gamma_1\theta; \Delta, \Delta_2\theta, \boxed{\forall(\varphi_{l_1} \otimes \varphi_{l_2} \multimap \varphi_P)} \Longrightarrow_\Sigma \psi}\,\forall\mathsf{L(repeated)}$$

$$\dfrac{\Gamma, \Gamma_1\theta; \Delta, \Delta_2\theta, \boxed{\forall(\varphi_{l_1} \otimes \varphi_{l_2} \multimap \varphi_P)} \Longrightarrow_\Sigma \psi}{\Gamma, \Gamma_1\theta; \Delta, \Delta_2\theta, \forall(\varphi_{l_1} \otimes \varphi_{l_2} \multimap \varphi_P) \Longrightarrow_\Sigma \psi}\,\mathsf{focusL}$$

Reusable $\mathcal{L}^1$ rules $!\forall(l \multimap P)$ are treated similarly. The correspondence is formalized in the following theorem and further discussed in Appendix B.3.

Theorem 4 (Soundness) *Assume that* $\vdash \Sigma.\langle \Omega \;;\; \Pi\rangle$ *state.*

1. *If* Π *is stable (i.e.,* Π *has the form* Π*) and* $\Sigma.\langle \Omega \;;\; \Pi\rangle \Mapsto \Sigma'.\langle \Omega' \;;\; \Pi'\rangle$*, then* $\lceil\Omega\rceil; \lceil\Pi\rceil \Longrightarrow_\Sigma \exists\Sigma'.\ulcorner !\Omega'; \Pi'\urcorner$.
2. *If* Π *is not stable and* $\Sigma.\langle \Omega \;;\; \Pi\rangle \Mapsto \Sigma'.\langle \Omega' \;;\; \Pi'\rangle$*, then* $\lceil\Omega\rceil; \lceil\Pi\rceil \Longrightarrow_\Sigma \exists\Sigma'.\ulcorner !\Omega'; \Pi'\urcorner$.

Although the connection between forward chaining languages and focused logic has been pointed out before, a precise account for a language as rich as $\mathcal{L}^1$ is new.

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 encoding of a generic $\mathcal{L}^1$ rule $\forall(l \multimap P)$ corresponds to such synthetic connectives, and its transitions map 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. In this section, we highlight the extension of $\mathcal{L}^1$ into $\mathcal{L}^{1.5}$.

Second-order variables, written X in $\mathcal{L}^{1.5}$, 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" [15]. 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). In fact, the extended language is no more expressive than $\mathcal{L}^1$. We rely on it as it will simplify our presentation.

The resulting mildly second-order language, $\mathcal{L}^{1.5}$, extends the grammar of $\mathcal{L}^1$ with the following productions:

$$\begin{array}{rrcl} \textit{Terms} & t & ::= & \ldots \mid X \mid p \\ \textit{Atoms} & A & ::= & \ldots \mid X\ t \\ \textit{Rules} & R & ::= & \ldots \mid \forall X : \tau \to o.\, R \\ \textit{Programs} & P & ::= & \ldots \mid \exists X : \tau \to o.\, P \end{array}$$

We upgrade the typing infrastructure of $\mathcal{L}^1$ by allowing second-order variables in the context Γ of the typing judgment, which is extended as follows:

$$\Gamma \quad ::= \quad \ldots \mid \Gamma, X : \tau \to o$$

The typing rules for each of the new syntactic forms are given next:

$$\ldots \qquad \frac{}{\Gamma, X : \tau \to o \vdash_\Sigma X : \tau \to o} \qquad\qquad \frac{}{\Gamma \vdash_{\Sigma, p:\tau\to o} p : \tau \to o}$$

$$\ldots \qquad \frac{\Gamma, X : \tau \to o \vdash_\Sigma t : \tau}{\Gamma, X : \tau \to o \vdash_\Sigma X\ t\ \mathit{atom}}$$

$$\ldots \qquad \frac{\Gamma, X : \tau \to o \vdash_\Sigma R\ \mathit{rule}}{\Gamma \vdash_\Sigma \forall X : \tau \to o.\, R\ \mathit{rule}}$$

$$\ldots \qquad \frac{\Gamma \vdash_{\Sigma, X:\tau\to o} P\ \mathit{prog}}{\Gamma \vdash_\Sigma \exists X : \tau \to o.\, P\ \mathit{prog}}$$

Just like for the first-order case, traversing a universal quantifier adds the bound variable to the context while type-checking an existential quantifier adds it to the signature. As in $\mathcal{L}^1$, we treat existential variables as if they were names.

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 that a free universal variable in a program position must appear in an earlier left-hand side. 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

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

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 has been passed to it in a "known" atom. The following rule is instead acceptable:

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

as only the predicate names marked to be deleted can trigger this rule. Appendix A.3 presents a variant of the type system of $\mathcal{L}^{1.5}$ that checks this safety constraint on second-order variables as well as our original safety constraint.

The rewriting semantics of $\mathcal{L}^{1.5}$ extends that of $\mathcal{L}^1$ slightly by processing the second-order quantifiers. The notion of state remains unchanged. The unfocused rewriting semantics is extended with the following two transitions:

$$\begin{array}{lcll} \Sigma.\langle \Omega\ ;\ \Pi, \forall X : \tau \to o.\, R\rangle & \mapsto & \Sigma.\langle \Omega\ ;\ \Pi, [p/X]R\rangle & \text{if } p : \tau \to o \text{ in } \Sigma \\ \Sigma.\langle \Omega\ ;\ \Pi, \exists X : \tau \to o.\, P\rangle & \mapsto & (\Sigma, X : \tau \to o).\langle \Omega\ ;\ \Pi, P\rangle & \end{array}$$

Observe that the first rule replaces the second-order order variable X with a predicate symbol from the signature.

The focused transition semantics undergoes a similar extension. It adds a rule to process second-order existential programs in a non-stable state

$$\Sigma.\langle \Omega\ ;\ \Pi, \exists X : \tau \to o.\, P\rangle \quad \Mapsto \quad (\Sigma, X : \tau \to o).\langle \Omega\ ;\ \Pi, P\rangle$$

while second-order universal quantifiers are handled in the transitions for rules in the archive and program portions of the state. These rules stay therefore as in Section 2.1.3.

The properties seen for $\mathcal{L}^1$, in particular type preservation, also hold for $\mathcal{L}^{1.5}$.

The second-order artifacts of $\mathcal{L}^{1.5}$ are not essential for our development, although they will simplify the discussion and make it more intuitive. In fact, any program in $\mathcal{L}^{1.5}$ can be transformed into a program in $\mathcal{L}^1$ that performs the same computation. We will now give a (slightly simplified) intuition of the workings of this transformation — a formal definition can be found in Appendix A.4.

Let P be a program in $\mathcal{L}^{1.5}$. For every type τ such that the second-order quantifications $\forall X : \tau \to o$ or $\exists X : \tau \to o$ appears in P, define the new predicate symbol $p_\tau : \iota \times \tau \to o$ and collect these predicates into a signature $\Sigma^{\cdot 5}$. Next, replace second-order quantifications and variable uses as follows:

- $\forall X : \tau \to o. \cdots X\ t \cdots (X) \cdots$ becomes $\forall x_X : \iota. \cdots p_\tau(x_X, t) \cdots (x_X) \cdots$
- $\exists X : \tau \to o. \cdots X\ t \cdots (X) \cdots$ becomes $\exists x_X : \iota \to \top. \cdots p_\tau(x_X\ (), t) \cdots (x_X\ ()) \cdots$

In both cases, all uses of the variable X as a predicate name are replaced by p_X as shown. Moreover, all occurrences of X in a term are replaced with the first-order variable x_X. Observe that universal and existential second-order variables are treated differently, due to the distinct typing patterns that universal and existential first-order variables have. For any predicate name p appearing in signature Σ of a state $\Sigma.\langle \Omega\ ;\ \Pi \rangle$ as $p : \tau \to o$, we define a new constant $x_p : \top \to \iota$ in $\Sigma^{.5}$, replace every occurrence of p in a term position in Π with $x_p\ ()$ and very atom $p\ t$ with $p_\tau(x_p\ (), t)$.

This encoding transforms valid states in $\mathcal{L}^{1.5}$ into valid states in $\mathcal{L}^{1}$ and maps transition steps in $\mathcal{L}^{1.5}$ to transition steps in $\mathcal{L}^{1}$. Appendix A.4 spells out the details of the transformation and a formal account of these results.

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 see how to write rules in a way as to emulate some characteristic or other of a module system, 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 indeed be automatically inferred from the way the variables they annotate are used.

3.1 Name Space Separation

Our first goal will be to allow rules, possibly executing concurrently, to make use of the functionalities provided by a program component without interfering with each other via this component. Those rules will be the client code and the program component will be the module. Key to avoiding interferences will be to give access to the module functionalities to each client through a different name.

As our first case study, consider the problem of adding two numbers written in unary — 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

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

In fact, 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.[1] 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 code snippet:

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

Another common usage pattern, more in the flavor of process algebraic specifications, is to embed the second rule in the right-hand side of the first:

$$\forall m\colon \mathsf{nat}.\, \forall n\colon \mathsf{nat}.\ \cdots \quad \multimap \quad \left[\begin{array}{rcl} \cdots\, , \mathsf{add}(n, m), & & \\ \forall r\colon \mathsf{nat}.\ \mathsf{add}(\mathsf{z}, r), \cdots & \multimap & \cdots \end{array}\right]$$

While this code achieves the desired effect in this case, it is incorrect whenever there is the possibility of two clients performing an addition at the same time. In fact, any concurrent execution making 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.

One way to obviate to this problem is to include an additional argument in the definition of add to disambiguate calls. The client code generates a unique identifier using existential quantification and retrieves the result that matches this identifier. The updated code for add and for the caller is as follows:

$$\begin{array}{rcl} !\forall x\colon \mathsf{nat}.\, \forall y\colon \mathsf{nat}.\, \forall id\colon \iota.\ \mathsf{add}(id, \mathsf{s}(x), y) & \multimap & \mathsf{add}(id, x, \mathsf{s}(y)) \\ \forall m\colon \mathsf{nat}.\, \forall n\colon \mathsf{nat}.\quad \cdots & \multimap & \exists id\colon \iota.\ \left[\begin{array}{rcl} \cdots\, , \mathsf{add}(id, n, m), & & \\ \forall r\colon \mathsf{nat}.\ \mathsf{add}(id, \mathsf{z}, r), \cdots & \multimap & \cdots \end{array}\right] \end{array}$$

While this intervention solves the above concern, it does not prevent a programmer from misusing the predicate add, either accidentally or intentionally. In fact, she can intercept results (or partial results) with rules of the form

$$\forall x\colon \iota.\, \forall m\colon \mathsf{nat}.\, \forall n\colon \mathsf{nat}.\, \mathsf{add}(x, m, n) \multimap \mathsf{add}(x, \mathsf{z}, \mathsf{s}(\mathsf{z}))$$

[1] This is not the only way to implement unary addition using rewrite rules and in fact not the best one. It is good for illustration however.

This returns the (typically incorrect) result $\mathsf{s}(\mathsf{z})$ to any legitimate caller of this adder. The problems stems from two facts: first the predicate symbol used internally in the implementation to perform the addition is publicly known, the second is that terms, even when existentially generated, are not private as they can be matched as just shown.

Both issues can be solved by using a second-order existential quantifier to generate the predicate used to carry out the addition. The following code snippet fully address the possibility of interferences. Observe that it does so without polluting the definition of addition with a third argument.

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

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

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

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 repeat the same piece of code (the inner reusable rule for addition above). Such code duplication goes against the spirit of modularity as it prevents code reuse and reduces maintainability. For definitions larger than this minimal case study, it also clutters the client code thereby diminishing readability.

One approach is to separate out the common code for addition and splice it back just before execution. This is essentially the solution advocated in [28, 27] as "elaboration". Achieving it within the model of computation of $\mathcal{L}^{1.5}$ would require significant, and probably ad-hoc, extensions.

A better approach is to provide a public name for the addition functionality, for example through the predicate name 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:

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

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 putting $add(\mathsf{z},r)$ in the left-hand side of an embedded rule like before.

This idea will be the cornerstone of our approach to adding modules to $\mathcal{L}^{1.5}$. Because it requires that the module and client code abide by a rather strict format, it will be convenient to provide the programmer with derived syntax. We will write the exported module (the first line in our last example) as follows:

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

Here, the public name adder is used as the name of the module. The names of the exported operations 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 via the atom $\mathsf{adder}(X)$ in the left-hand side of a rule. 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. The second rule of our example is re-expressed as follows:

$$\forall m{:}\,\mathsf{nat}.\,\forall n{:}\,\mathsf{nat}.\ \cdots \quad \multimap \quad A\ \mathbf{as}\ \mathsf{adder}.\ \left[\begin{array}{l} \cdots, A.add(n,m), \\ \forall r{:}\,\mathsf{nat}.\ A.add(\mathsf{z},r), \cdots \quad \multimap \quad \cdots \end{array}\right]$$

Here, A is a *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 allow using the exact same names exported by a module in its **provide** stanza. Uses of compound names and the **as** construct are elaborated into our original code, as we will see in Section 4.

The **provide** stanza defines the functionalities exported by a module. We can collect these declarations and give them a name in an *interface* for the module, which corresponds to the notion of signature in Standard ML. Our ongoing example leads to the following interface declaration:

interface ADDER
  add : $\mathsf{nat} \times \mathsf{nat} \to o$
end

This gives us a way to provide a name for this interface, here ADDER. The module adder can then be explicitly annotated with the name of its interface:

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

As in SML, we could use such annotations to restrict or even replace the **provide** stanza in the module definition. We will not explore this possibility further. Interfaces will come handy later. Until then, we will not bother with them.

To summarize what we have achieved so far, we have devised a template to support basic modularity in $\mathcal{L}^{1.5}$ and we have developed some derived syntax for it. The same module can be used by multiple clients at the same time without the danger of (intentional or accidental) interferences. Multiple implementations for the same interface are supported, although our example is still too simple to demonstrate this.

3.2 Modes

In the face of it, our adder module is a bit strange as 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. We can write this directly in $\mathcal{L}^{1.5}$ as the follows:

$$
\begin{array}{rl}
!\forall add\_req\colon \mathsf{nat} \times \mathsf{nat} \to o. & \\
\quad \forall add\_res\colon \mathsf{nat} \to o. & \mathsf{adder}'(add\_req, add\_res) \quad \multimap \\
& \quad \left[\begin{array}{rcl} !\forall x\colon \mathsf{nat}.\, \forall y\colon \mathsf{nat}. & add\_req(\mathsf{s}(x), y) & \multimap \quad add\_req(x, \mathsf{s}(y)) \\ \forall z\colon \mathsf{nat}. & add\_req(\mathsf{z}, z) & \multimap \quad add\_res(z) \end{array}\right] \\
\forall m\colon \mathsf{nat}.\, \forall n\colon \mathsf{nat}. & \cdots \quad \multimap \\
& \quad \exists add\_req\colon \mathsf{nat} \times \mathsf{nat} \to o. \\
& \quad \exists add\_res\colon \mathsf{nat} \to o. \left[\begin{array}{l} \mathsf{adder}'(add\_req, add\_res), \\ \cdots, add\_req(n, m), \\ \forall r\colon \mathsf{nat}.\ add\_res(r), \cdots \quad \multimap \quad \cdots \end{array}\right]
\end{array}
$$

Now, 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 (and interfaces) with *modes* that describe how exported predicates are to be used. The mode **in** in the **provide** stanza is used to check that a (compound) predicate is used only on the left-hand side of a rule, while the mode **out** forces usage on the right-hand side only. A declaration without such a marker, for example add in the last section, can be used freely. Our new adder module is written as follows in the concrete syntax:

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

and the client code assumes the following concrete form:

$$
\forall m\colon \mathsf{nat}.\, \forall n\colon \mathsf{nat}.\ \cdots \quad \multimap \quad A\ \textbf{as}\ \mathsf{adder}'. \left[\begin{array}{l} \cdots, A.add\_req(n, m), \\ \forall r\colon \mathsf{nat}.\ A.add\_res(r), \cdots \quad \multimap \quad \cdots \end{array}\right]
$$

3.3 Abstract Data Types

The infrastructure we developed so far already allows us to write modules that implement abstract data types. For example, the following module (in concrete syntax) implements a simple dictionary with three operations: lookup (split into a request and a result predicate name), insert and delete. Keys are natural numbers (of type nat) and values are terms (of type ι).

$$
\begin{array}{l}
\textbf{module}\ \mathsf{dictionary} \\
\quad \textbf{provide}\quad \textbf{out}\ \ lookup\_req : \mathsf{nat} \to o \qquad \textbf{in}\ \ lookup\_res : \mathsf{nat} \times \iota \to o \\
\qquad\qquad\ \ \textbf{out}\ \ \ insert : \mathsf{nat} \times \iota \to o \\
\qquad\qquad\ \ \textbf{out}\ \ \ delete : \mathsf{nat} \to o \\
\quad \textbf{local}\quad data\ \ : \mathsf{nat} \times \iota \to o \\
\\
\quad\ !\forall k\colon \mathsf{nat}.\, \forall v\colon \iota.\ \ insert(k, v) \quad \multimap \quad data(k, v) \\
\quad\ !\forall k\colon \mathsf{nat}.\, \forall v\colon \iota.\ \ lookup\_req(k), data(k, v) \quad \multimap \quad lookup\_res(k, v), data(k, v) \\
\quad\ !\forall k\colon \mathsf{nat}.\, \forall v\colon \iota.\ \ delete(k), data(k, v) \quad \multimap \quad \cdot \\
\textbf{end}
\end{array}
$$

The **local** stanza introduces a name that is used locally within the module. It is elaborated into an existential surrounding the rules defining the module. The $\mathcal{L}^{1.5}$ program corresponding to this module is as follows:

$$\forall lookup\_req\colon \mathsf{nat} \to o.\ \forall lookup\_res\colon \mathsf{nat} \times \iota \to o.\ \forall insert\colon \mathsf{nat} \times \iota \to o.\ \forall delete\colon \mathsf{nat} \to o.$$
$$\mathsf{dictionary}(lookup\_req, lookup\_res, insert, delete) \quad \multimap \quad \exists data\colon \mathsf{nat} \times \iota \to o.$$
$$\left[\begin{array}{lll} !\forall k\colon \mathsf{nat}.\ \forall v\colon \iota. & insert(k,v) & \multimap \quad data(k,v) \\ !\forall k\colon \mathsf{nat}.\ \forall v\colon \iota. & lookup\_req(k), data(k,v) & \multimap \quad lookup\_res(k,v), data(k,v) \\ !\forall k\colon \mathsf{nat}.\ \forall v\colon \iota. & delete(k), data(k,v) & \multimap \quad \cdot \end{array}\right]$$

As a more complex example of abstract data type, the following module implements stacks of natural numbers. Internally, stacks are linked lists built from the local predicates $head(d'')$, $elem(n, d', d'')$ and $empty(d')$. Here, the terms d' and d'' implement the links of the linked list by identifying the predecessor and the successor element, respectively. Such a list will start with the atom $head(d_0)$, continue as series of atoms $elem(n_i, d_i, d_{i+1})$ and end with the atom $empty(d_n)$. The code maintains that invariant that links are used exactly this way.

module stack
  provide **out** $push : \mathsf{nat} \to o$
  out $pop\_req : o$ **in** $pop\_res : \mathsf{nat} \to o$
  out $isempty : o$
  out $size\_req : o$ **in** $size\_res : \mathsf{nat} \to o$

  local $empty : \iota \to o$
  $elem : \mathsf{nat} \times \iota \times \iota \to o$
  $head : \mathsf{nat} \times \iota \to o$

$$\cdot \quad \multimap \quad \exists d\colon \iota.\ empty(d), head(d)$$
$$!\forall e\colon \mathsf{nat}.\ push(e), head(d) \quad \multimap \quad \exists d'\colon \iota.\ elem(e, d', d), head(d')$$
$$!\forall e\colon \mathsf{nat}.\ \forall d\colon \iota.\ \forall d'\colon \iota.\ pop\_req, head(d), elem(e, d, d') \quad \multimap \quad pop\_res(e), head(d')$$
$$!\forall d\colon \iota.\ isempty, head(d), empty(d) \quad \multimap \quad head(d), empty(d)$$
$$!\forall d\colon \iota.\ size\_req, head(d) \quad \multimap \quad \exists size'\colon \mathsf{nat} \times \iota \to o.\ head(d), size'(\mathsf{z}, d),$$
$$\left[\begin{array}{lll} !\forall n\colon \mathsf{nat}.\ \forall e\colon \mathsf{nat}.\ \forall d\colon \iota.\ \forall d'\colon \iota. & size'(n,d), elem(e,d,d') & \multimap \quad elem(e,d,d'), size'(\mathsf{s}(n), d') \\ \forall n\colon \mathsf{nat}.\ \forall d\colon \iota. & size'(n,d), empty(d) & \multimap \quad empty(d), size\_res(n) \end{array}\right]$$
end

3.4 Parametric Modules – I

Our next challenge will be to implement a module that provides a functionality to increment a number by n, where n is a parameter passed when the module is instantiated in the client code. This corresponds to a Standard ML functor that takes a value as an argument.

Such parametricity can be achieved by simply passing n as an argument to the public predicate used to call the module, in addition to the predicates for each declaration in its **provide** stanza. In concrete syntax, we can simply give n as an argument in our module declaration. Here is an implementation:

$$
\begin{array}{l}
\textbf{module}\ \mathsf{increment}\ (n\colon \mathsf{nat}) \\
\quad \textbf{provide}\quad \textbf{out}\ \ \mathit{incr\_req} : \mathsf{nat} \to o \quad \textbf{in}\ \ \mathit{incr\_res} : \mathsf{nat} \to o \\
\quad \textbf{local}\quad \mathit{add}\quad : \mathsf{nat} \times \mathsf{nat} \to o \\
\qquad\qquad\qquad\qquad \begin{array}{rrrl}
\forall m\colon \mathsf{nat}. & \mathit{incr\_req}(m) & \multimap & \mathit{add}(m, n) \\
!\forall m\colon \mathsf{nat}.\, \forall n\colon \mathsf{nat}. & \mathit{add}(\mathsf{s}(m), n) & \multimap & \mathit{add}(m, \mathsf{s}(n)) \\
\forall r\colon \mathsf{nat}. & \mathit{add}(\mathsf{z}, r) & \multimap & \mathit{incr\_res}(r)
\end{array} \\
\textbf{end}
\end{array}
$$

The value m passed as a parameter through the requester $\mathit{incr\_req}(m)$ is added to n using the same code we wrote for our adder. The result is returned through the predicate $\mathit{incr\_res}$. The above module is written as follows in $\mathcal{L}^{1.5}$:

$$
\begin{array}{l}
\forall n\colon \mathsf{nat}.\, \forall \mathit{incr\_req}\colon \mathsf{nat} \to o.\, \forall \mathit{incr\_res}\colon \mathsf{nat} \to o.\ \mathsf{increment}(n, \mathit{incr\_req}, \mathit{incr\_res}) \quad \multimap \\
\qquad\qquad \exists \mathit{add}\colon \mathsf{nat} \times \mathsf{nat} \to o.\ \left[\begin{array}{rrrl}
\forall m\colon \mathsf{nat}. & \mathit{incr\_req}(m) & \multimap & \mathit{add}(m, n) \\
!\forall m\colon \mathsf{nat}.\, \forall n\colon \mathsf{nat}. & \mathit{add}(\mathsf{s}(m), n) & \multimap & \mathit{add}(m, \mathsf{s}(n)) \\
\forall r\colon \mathsf{nat}. & \mathit{add}(\mathsf{z}, r) & \multimap & \mathit{incr\_res}(r)
\end{array} \right]
\end{array}
$$

Although plausible, this code is incorrect as two invocations of $\mathit{incr\_req}$ within the same left-hand side may cause internal interferences as the local predicate symbol add is shared: the two calls would not be able to tell which result is which. The following code corrects this mistake by making add local to each use of $\mathit{incr\_req}$.

$$
\begin{array}{l}
\textbf{module}\ \mathsf{increment}\ (n\colon \mathsf{nat}) \\
\quad \textbf{provide}\quad \textbf{out}\ \ \mathit{incr\_req} : \mathsf{nat} \to o \quad \textbf{in}\ \ \mathit{incr\_res} : \mathsf{nat} \to o \\
\quad \forall m\colon \mathsf{nat}.\ \mathit{incr\_req}(m) \quad \multimap \quad \exists \mathit{add}\colon \mathsf{nat} \times \mathsf{nat} \to o. \\
\qquad\qquad\qquad\qquad \left[\begin{array}{rrrl}
\mathit{add}(m, n) & & & \\
!\forall m\colon \mathsf{nat}.\, \forall n\colon \mathsf{nat}. & \mathit{add}(\mathsf{s}(m), n) & \multimap & \mathit{add}(m, \mathsf{s}(n)) \\
\forall r\colon \mathsf{nat}. & \mathit{add}(\mathsf{z}, r) & \multimap & \mathit{incr\_res}(r)
\end{array} \right] \\
\textbf{end}
\end{array}
$$

While correct, this code is a bit silly as it re-implements our adder. An even better solution is use the adder' module to perform the addition, as done here:

$$
\begin{array}{l}
\textbf{module}\ \mathsf{increment}\ (n\colon \mathsf{nat}) \\
\quad \textbf{provide}\quad \textbf{out}\ \ \mathit{incr\_req} : \mathsf{nat} \to o \quad \textbf{in}\ \ \mathit{incr\_res} : \mathsf{nat} \to o \\
\quad \forall m\colon \mathsf{nat}.\ \mathit{incr\_req}(m) \quad \multimap \quad A\ \textbf{as}\ \mathsf{adder}'.\ \left[\begin{array}{l}
A.\mathit{add\_req}(m, n), \\
\forall r\colon \mathsf{nat}.\ A.\mathit{add\_res}(r) \quad \multimap \quad \mathit{incr\_res}(r)
\end{array} \right] \\
\textbf{end}
\end{array}
$$

Modules can therefore call each other.

Now, client code would make use of our $\mathsf{increment}$ module as follows:

$$
\forall m\colon \mathsf{nat}.\, \forall n\colon \mathsf{nat}.\ \cdots \quad \multimap \quad I\ \textbf{as}\ \mathsf{increment}(n).\ \left[\begin{array}{l}
\cdots, I.\mathit{incr\_req}(m), \\
\forall r\colon \mathsf{nat}.\ I.\mathit{incr\_res}(r), \cdots \quad \multimap \quad \cdots
\end{array} \right]
$$

3.5 Sharing Private Names

As our next example, we will implement — and use — a module that provides the functionalities of a reference cell in a stateful language. This cell is initialized as we instantiate the module (using the approach to parametrization seen in the last section) and provides private predicates to get the current value of the cell and to set it to a new value. At this stage, this is easily done by the following code, which relies on the private predicate *content* to store the content of the cell.

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

Differently from the example in the last section, the fact that *content* is not local to each invocation of *get* and *set* is a feature as we want all uses of the cell to refer to the same content.

Reference cells are often shared by various subcomputations. One way to do so is to pass the exported predicates that manipulate them 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 getter to the third rule through the predicate q. The second rule can only write to the cell (it replaces its content with z in this example). The third rule can only read the content of the cell (here, it then outputs it through predicate s).

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

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

Note that reference cells can be implemented in a much simpler way by exporting the predicate that holds the content of the cell. This is reminiscent of what we did with our first adder. The code is as follows:

$$
\begin{array}{l}
\textbf{module}\ \mathsf{cell}'\ (v\colon \mathsf{nat}) \\
\quad \textbf{provide}\quad \mathit{content} : \mathsf{nat} \to o \\
\quad\ \cdot\ \multimap\ \mathit{content}(v) \\
\textbf{end}
\end{array}
$$

Now, however, there is no way for the client code to pass distinct capabilities to subcomputations. In the following code, the first rule passes the compound predicate name $C.\mathit{content}$ to the second and the third rule, thereby giving both of them read and write access to the cell.

$$
\begin{array}{rcl}
\cdot & \multimap & C \ \mathbf{as}\ \mathsf{cell}'(\mathsf{s}(\mathsf{z})).\ \left[\begin{array}{l} p(C.content), \\ q(C.content) \end{array}\right] \\
\forall c\colon \mathsf{nat} \to o.\ \forall v\colon \mathsf{nat}.\ \left[\begin{array}{l} p(c), \\ c(v) \end{array}\right] & \multimap & c(\mathsf{z}) \\
\forall c\colon \mathsf{nat} \to o.\ \forall v\colon \mathsf{nat}.\ \left[\begin{array}{l} q(c), \\ c(v) \end{array}\right] & \multimap & s(v)
\end{array}
$$

3.6 Parametric Modules – II

As our last, and most complex, example, we implement two modules for queues. We then use either module to instantiate a buffer for a producer and a consumer to exchange data.

Queues have just 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. These operations are specified by the following interface:

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

We give two implementations of queues. The first one, which we call queue, uses linked lists and is similar to stacks earlier.

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

Our second implementation relies on a term representation of lists, with private constructors nil and $cons$.

$$
\begin{array}{l}
\textbf{module}\ \mathsf{queue}'\colon \mathsf{QUEUE} \\
\quad \textbf{provide}\quad \textbf{out}\quad enq : \mathsf{nat} \to o \\
\qquad\qquad\ \ \textbf{out}\quad deq\_req : o \qquad\quad \textbf{in}\quad deq : \mathsf{nat} \to o \\
\\
\quad \textbf{local}\quad nil\ : \iota \\
\qquad\qquad cons\ : \mathsf{nat} \times \iota \to \iota \\
\qquad\qquad\ \ q\ : \iota \to o \\
\qquad\qquad\qquad\qquad\qquad \cdot \quad \multimap \quad q(nil) \\
\quad !\forall e\colon \mathsf{nat}.\, \forall tail\colon \iota.\ \ enq(e), q(tail) \quad \multimap \quad q(cons(e, tail)) \\
\\
\qquad\qquad !\forall e\colon \mathsf{nat}.\ \ deq\_req, q(tail) \quad \multimap \\
\qquad\qquad\qquad\qquad\qquad \left[\begin{array}{l} q(tail), \\ q(last(tail)), \\ \qquad \forall v\colon \mathsf{nat}. \qquad last(cons(v, nil)) \qquad \multimap \quad deq(v) \\ ! \left[\begin{array}{l} \forall v\colon \mathsf{nat}. \\ \forall v'\colon \mathsf{nat}. \\ \forall tail\colon \iota. \end{array}\right] \ last(cons(v, cons(v', tail))) \quad \multimap \quad last(cons(v', tail)) \end{array}\right] \\
\qquad\qquad\qquad\quad \exists last\colon \iota \to o. \\
\textbf{end}
\end{array}
$$

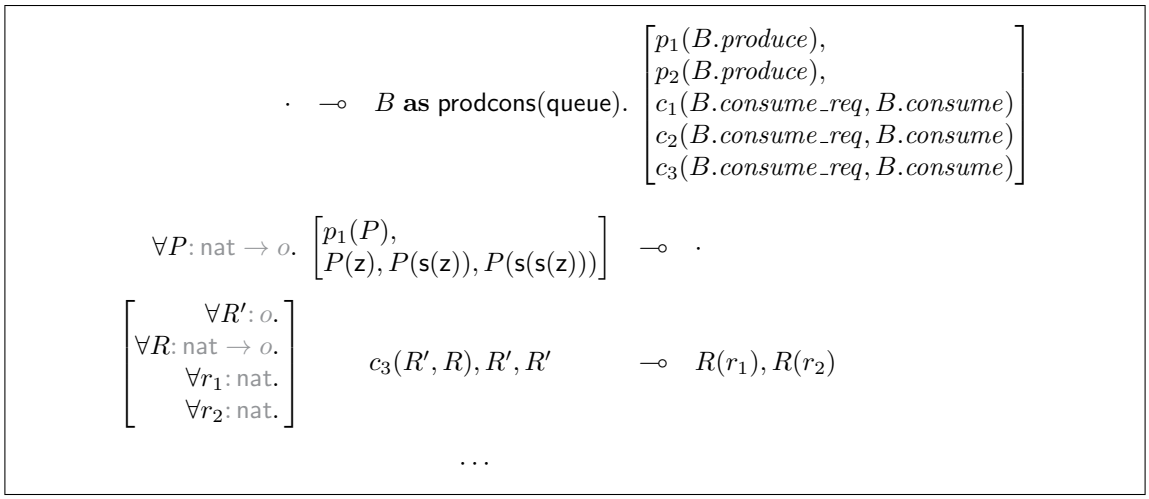
Observe that both implementations ascribe to the same interface for queues, QUEUE.

Next, we define a producer-consumer module that exports predicates to produce a natural number or to consume it. In this module, we use a queue as the buffer where the producer deposits data and the consumer retrieves them from — in truth, our producer-consumer module does nothing more than map these operations to the underlying enqueuing and dequeuing facilities.

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

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

The argument Q of prodcons is the name of a module with interface QUEUE, which gives use a choice between queue and queue$'$. The following code chooses queue (arbitrarily), and triggers two producers and three consumers by passing them the appropriate predicates exported by the module. We also show one such producer (triggered by predicate p_1) which pushes three integers in the buffer, and one consumer (c_3) that retrieves two elements.



This example showed how modules can take other modules as arguments, just like Standard ML's functors. In fact, our language can do a lot more. 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 (of which we showed just one). 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 [29]). 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 some convenience syntax for the programming patterns of $\mathcal{L}^{1.5}$ that realized module definitions ($\mathbf{module}\ \ldots\ \mathbf{end}$), module instantiations ($N\ \mathbf{as}\ p\,t.\ \ldots$), and the use of exported names (e.g., $N.add$).

2. We imposed a series of restrictions on where and how predicates could be used (**in** for left-hand side only, **out** for right-hand side only), 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 in (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 all the ingredients to write modular code already. We call this process *elaboration*. We handle the restrictions in (2) by typechecking $\mathcal{L}^{M}$ code, as a user could violate them even if her code elaborated to a valid $\mathcal{L}^{1.5}$ program. Appendix A.5 describes an extension of the typing rules of Section 2.1.2 that checks that these restrictions as satisfied at the level of $\mathcal{L}^{M}$. It also summarizes the syntax of our module language (which 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 the native syntax of $\mathcal{L}^{1.5}$. We then execute this $\mathcal{L}^{1.5}$ program. We will now outline how to elaborate away the two added constructs, **module** and **end**. See Appendix A.5 for a precise description.

Recall the general syntax of modules definition has the form:

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

What we called the module interface consists of the signature Σ_{export} on the **provide** stanza. 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 following $\mathcal{L}^{1.5}$ 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 $\exists\Sigma.\,P$.

Next, we handle module instantiation, whose derived syntax, is

$$N\ \mathbf{as}\ p\,t.\,P$$

Let Σ_{export} be the interface exported by the module for p, defined as in the previous paragraph. It will be convenient to express Σ_{export} in the form $\vec{X} : \vec{\tau}$, where the i-th declaration in Σ_{export} is $X_i : \tau_i$. Then, we elaborate the above construct as

$$\exists\Sigma_{export}.\ \left[\begin{array}{l} p(t, \vec{X}) \\ [\vec{X}/N.\vec{X}]P \end{array}\right]$$

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, as just described, 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 Related Work

Programming languages equipped with a module system started appearing in the early 1970s, with Modula-2 [37] and Standard ML [29] still in use today. Modern languages, from Java to JavaScript, invariably feature a module system of some sort. The module system of ML [25, 29] is still considered by many the gold standard because of it support for parametric modules (functors) and fine control on exported functionalities. Extensions, in particular with higher-order modules are active areas of research. It should be noted however that large amounts of code are still written nowadays in non-modular languages, in particular in C.

Module research in logic programming has largely targeted languages based on backward search. See [4] for a comprehensive survey as of the mid-1990s. Several commercial and open-source implementations of Prolog feature a module system, for example SICStus Prolog and SWI Prolog. Declarative approaches to modularity have been investigated in relation to extensions of Prolog. In particular [27, 28] bases the module system for λProlog on this language's support for embedded implication and universal quantification, in a fashion that resembles our use of existential quantifiers. Modules are however dynamically spliced into the body of each clause that uses them, and therefore are not logically separate entities. Abdallah [1] proposes using second-order variables as a declarative support for procedure calls in a Prolog-like language, achieving a form of modularity along the way. This is closely related to our use of second-order quantification and to the logic recently investigated in [15]. Several proposals have been made for introducing modules in dependently-typed backward-chaining languages, for example [32] for LF.

Module systems for languages based on forward logic programming have been few and far between. Cervesato proposed an ad-hoc module system for the multiset rewriting language MSR [6]. Some languages based on general rewriting have been extended with a module system, for example Maude [26, 10]. Process algebras [34, 13] still tend to be used for specification rather than for general programming, yet some research has been done about modularizing them, especially for assurance purposes [33].

The relationship between logic and rule-based languages has been explored in great depth. Most relevant to the present work is [9] that shows how the left sequent rules of most connectives of linear logic yield a rewriting behavior that subsumes traditional multiset rewriting and established models of concurrency such as Petri nets [30] as well as many process algebras. The unfocused semantics of $\mathcal{L}^{1.5}$ discussed in this report is largely based on a subset of these operators. The connection between focusing and forward chaining has also been the subject of recent study, for example in [35].

Forward-chaining logic programming gained prominence in the 1980s with Datalog [16, 5] in the database area. This paradigm has been having a renaissance of late and has been the basis for languages in the domains of security and trust management [23], networking [18, 24], robot coordination [2], multicore programming [11], graph programming [12] and concurrent specifications [36] just to cite a few. It is also a foundation for the general purpose programming language CHR [14] and earlier GAMMA [3] as well as CoMingle, a language for mobile application development [21].

6 Future Work and Conclusions

In this report, we developed an advanced module system for $\mathcal{L}^1$, a small rule-based programming language. This language 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, which means that not only do they too have a natural interpretation in logic, but they are also first-class entities in our language. They make use of a mild form of second-order quantification, which can be compiled to first-order quantification but at the cost of complicating somewhat the static semantics of our language. Although our module infrastructure is nothing more than normal rules used in a specific way, they share many of the features found in advanced module systems such as that of Standard ML [25]. In particular, they enable private names, which in turn support abstract data types, name space separation and other forms of information hiding. Our modules can be invoked with parameters, both values supplied by a client rule as well as other modules. This corresponds directly to the notion of functor in Standard ML. Being first-class, higher-order modules and recursive functors are natively supported. The underlying rewriting model also enables sharing private names across threads of computation, which is critical for the kind of concurrent settings multiset rewriting is applied most frequently nowadays [20].

The rewriting language underlying our modular scaffolding is of interest in its own right. Similarly to the language of ω-multisets [9], it can be viewed as a specific reading of the traditional inference rules for a fragment of linear logic. It subsumes most multiset rewriting formalisms we are aware of (e.g., [14, 36]), forward logic programming languages such as Datalog [16, 5] and recent refinements [18, 24, 2, 11, 12], and many process algebras [34, 13]. This entails that the module infrastructure developed in this report can be applied directly to any of these formalisms.

Our immediate next step will be to implement our module system in the CoMingle system [21]. CoMingle is a logic programming framework aimed at simplifying the development of applications distributed over multiple Android devices. It currently relies on the fragment of the language discussed in this report deprived of nested rules. It however extends this language with multiset comprehensions, which allow atomically manipulating arbitrarily many atoms that match a pattern, and provides a declarative interface to the underlying Android runtime system. 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 extend the core language we defined in this report. Here, second-order variables stood for predicate names only, making them more like the quantifiers of Gabbay and Mathijssen's "one-and-a-halfth-order logic" [15]. Although matching in second-order logic is decidable, it is expensive in its most general form [19]. However, second-order matching also supports features such as reflection and the runtime construction of rules, which are valuable programming tools. We want to investigate relaxing our current constraint while preserving performance. Our core language also possesses numerous features found in process algebras, in particular nested actions and the dynamic creation of names. We expect many of the verification techniques developed for process algebras, for example bisimilarity and congruence in the π-calculus [34], can be adapted to our language. This would be particularly interesting as reasoning about multiset rewriting is usually based on a different set of techniques, often reachability arguments.

References

[1] M.A. Nait Abdallah. Procedures in Horn-Clause Programming. In Ehud Shapiro, editor, *Third International Conference on Logic Programming — ICLP'86*, pages 433–447. Springer LNCS 225, 1986.

[2] Michael P. Ashley-Rollman, Peter Lee, Seth Copen Goldstein, Padmanabhan Pillai, and Jason D. Campbell. A Language for Large Ensembles of Independently Executing Nodes. In *25th International Conference on Logic Programming — ICLP'09*, pages 265–280. Springer LNCS 5649, 2009.

[3] Jean-Pierre Banâtre and Daniel Le Métayer. Programming by Multiset Transformation. *Communications of the ACM*, 36(1):98–111, 1993.

[4] Michele Bugliesi, Evelina Lamma, and Paola Mello. Modularity in Logic Programming. *The Journal of Logic Programming*, 19–20(1):443–502, 1994.

[5] Stefano Ceri, Georg Gottlob, and Letizia Tanca. What You Always Wanted to Know About Datalog (And Never Dared to Ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, March 1989.

[6] Iliano Cervesato. MSR 2.0: Language Definition and Programming Environment. Technical Report CMU-CS-11-141, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, November 2011.

[7] Iliano Cervesato. Typed Multiset Rewriting Specifications of Security Protocols. Technical Report CMU-CS-11-140, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, October 2011.

[8] Iliano Cervesato, Frank Pfenning, David Walker, and Kevin Watkins. A Concurrent Logical Framework II: Examples and Applications. Technical Report CMU-CS-02-102, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 2003.

[9] Iliano Cervesato and Andre Scedrov. Relating State-Based and Process-Based Concurrency through Linear Logic. *Information & Computation*, 207(10):1044–1077, 2009.

[10] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. System Modules. In *All About Maude — A High-Performance Logical Framework*, pages 131–157. Springer LNCS 4350, 2007.

[11] Flávio Cruz, Michael P. Ashley-Rollman, Seth Copen Goldstein Goldstein, Ricardo Rocha, and Frank Pfenning. Bottom-Up Logic Programming for Multicores. In Vítor Santos Costa, editor, *7th International Workshop on Declarative Aspects and Applications of Multicore Programming — DAMP'12*. ACM Digital Library, January 2012. Short paper.

[12] Flávio Cruz, Ricardo Rocha, Seth Copen Goldstein Goldstein, and Frank Pfenning. A Linear Logic Programming Language for Concurrent Programming over Graph Structures. In *30th International Conference on Logic Programming — ICLP'14*, Vienna, Austria, 2014.

[13] Cédric Fournet. *The Join-Calculus: a Calculus for Distributed Mobile Programming*. PhD thesis, École Polytechnique, Palaiseau, 1998. INRIA TU-0556. Also available from `http://research.microsoft.com/~fournet`.

[14] Thom Frühwirth. *Constraint Handling Rules*. Cambridge University Press, 2009.

[15] Murdoch J. Gabbay and Aad Mathijssen. One-and-a-halfth-order Logic. *Journal of Logic and Computation*, 18(4):521–562, August 2008.

[16] Hervé Gallaire, Jack Minker, and Jean-Marie Nicolas. Logic and Databases: A Deductive Approach. *ACM Computing Survey*, 16(2):153–185, June 1984.

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

[18] Stéphane Grumbach and Fang Wang. Netlog, a Rule-based Language for Distributed Programming. In *12th International Conference on Practical Aspects of Declarative Languages — PADL'10*, pages 88–103. Springer LNCS 5937, 2010.

[19] Kouichi Hirata, Keizo Yamada, and Masateru Harao. Tractable and Intractable Second-order Matching Problems. *Journal of Symbolic Computation*, 37(5):611–628, 2004.

[20] Edmund S.L. Lam and Iliano Cervesato. Optimized Compilation of Multiset Rewriting with Comprehensions. In Jacque Garrigue, editor, *12th Asian Symposium on Programming Languages and Systems — APLAS'14*, pages 19–38, Singapore, 2014. Springer LNCS 8858.

[21] Edmund S.L. Lam, Iliano Cervesato, and Nabeeha Fatima Haque. Comingle: Distributed Logic Programming for Decentralized Mobile Ensembles. In Tom Holvoet and Mirko Viroli, editors, *17th IFIP International Conference on Coordination Models and Languages — COORDINATION'15*, pages 51–66, Grenoble, France, 2–4 June 2015. Inria, Springer LNCS 9037.

[22] Olivier Laurent, Myriam Quatrini, and Lorenzo Tortora de Falco. Polarized and Focalized Linear and Classical Proofs. *Annals of Pure and Applied Logic*, 134(2–3):217–264, July 2005.

[23] Ninghui Li and John C. Mitchell. Datalog with Constraints: A Foundation for Trust-management Languages. In *5th International Conference on Practical Aspects of Declarative Languages — PADL'03*, pages 58–73. Springer LNCS 2562, 2003.

[24] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative Networking: Language, Execution and Optimization. In *2006 ACM SIGMOD International Conference on Management Of Data — SIGMOD'06*, pages 97–108. ACM, 2006.

[25] David MacQueen. Modules for Standard ML. In *1984 ACM Symposium on LISP and Functional Programming Languages*, pages 198–207, 1984.

[26] José Meseguer and Christiano Braga. Modular Rewriting Semantics of Programming Languages. In Charles Rattray, Savitri Maharaj, and Carron Shankland, editors, *10th International Conference on Algebraic Methodology and Software Technology — AMAST'04*, pages 364–378. Springer LNCS 3116, 2004.

[27] Dale Miller. A Logical Analysis of Modules in Logic Programming. *Journal of Logic Programming*, 6(1-2):79–108, January 1989.

[28] Dale Miller. A Proposal for Modules in λProlog. In Roy Dyckhoff, editor, *4th Workshop on Extensions to Logic Programming — ELP'94*, pages 206–221. Springer LNCS 798, 1994.

[29] Robin Milner, Robert Harper, David MacQueen, and Mads Tofte. *The Definition of Standard ML – Revised*. MIT Press, 1997.

[30] Carl Adam Petri. Fundamentals of a Theory of Asynchronous Information Flow. In *Proceedings of IFIP Congress 62 on Information Processing*, pages 386–390, Munich, Germany, 1963. North Holland.

[31] Frank Pfenning. Structural Cut Elimination I. Intuitionistic and Classical Logic. *Information and Computation*, 157(1/2):84–141, 2000.

[32] Florian Rabe and Carsten Schürmann. A Practical Module System for LF. In James Cheney and Amy Felty, editors, *Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice — LFMTP'09*, pages 40–48, Montreal, Canada, 2009.

[33] Sriram Rajamani and Jakob Rehof. A Behavioral Module System for the Pi-Calculus. In Patrick Cousot, editor, *8th International Symposium on Static Analysis — SAS'01*, pages 375–394. Springer LNCS 2126, 2001.

[34] Davide Sangiorgi and David Walker. *The Pi-Calculus — a Theory of Mobile Processes*. Cambridge University Press, 2001.

[35] Robert J. Simmons. Structural Focalization. *ACM Transaction in Computational Logic*, 15(3):1–33, September 2014.

[36] Kevin Watkins, Frank Pfenning, David Walker, and Iliano Cervesato. Specifying Properties of Concurrent Computations in CLF. In Carsten Schürmann, editor, *Fourth Workshop on Logical Frameworks and Meta-languages — LFM'04*, pages 67–87, Cork, Ireland, 2007. ENTCS 199.

[37] Niklaus Wirth. *Programming in Modula-2*. Springer, 4th edition edition, 1988.

A Language Summary

This appendix summarizes the syntax (Appendix A.1), typing and operational semantics of the language $\mathcal{L}^{1.5}$ developed in this report. We collect the base typing rules of the language, deprived of safety checks, in Appendix A.2 and enrich them with such checks in Appendix A.3. We show how to translate $\mathcal{L}^{1.5}$ programs back to $\mathcal{L}^{1}$ in Appendix A.4. The syntax of modular language $\mathcal{L}^{M}$ is collected in Appendix A.5 together with typing rules for it. The summary of the operational semantics of our language starts in Appendix A.6 with a discussion of the underlying notion of congruence, continues in Appendix A.7 with an abstract presentation of its abstract rewriting semantics read off directly from the rules of the logic operator associated with each operator of $\mathcal{L}^{1.5}$ (see Appendix B), and ends with an operational semantics which is based on a focused presentation of this logic. Such an operational semantics is an essential step towards a practical implementation of this rewriting language.

A.1 Syntax

This appendix summarizes the syntax of the our core language $\mathcal{L}^{1.5}$. The addition that $\mathcal{L}^{1.5}$ brings about with respect to $\mathcal{L}^{1}$ are highlighted in blue.

The grammar below refers to the following syntactic categories, which are assumed given and distinguishable:

Term variables	x
Predicate variables	X
Function symbols	f
Predicate symbols	p

In the typing and evaluation semantics below, we blur the distinction between (term and predicate) variables introduced by an existential quantifier and (function and predicate) symbols. A more precise definition would instantiate such variables with new symbols when they are used, but at the cost of notational clutter — we prefer a simpler albeit slightly imprecise notation to a fully formal description.

The other entities of our language are defined by the grammar in Figure A.1.

In examples, we abbreviated the type $\top \to \iota$ as simply ι. This is the type of a function symbol that does not take any (proper) argument — a constant. Any signature declaration $f : \iota$ is therefore understood as $f : \top \to \iota$ and every term written f stands for $f\,()$. Similarly, we write the type $\top \to o$ as o — this is the type predicate names and variables without (proper) argument. A declaration $p : o$ stands therefore for $p : \top \to o$ and an atom p is shorthand for $p\,()$ — these are the propositional atoms.

Types	τ	$::=$	ι	individuals
		$\mid$	o	state objects
		$\mid$	$\tau \to \tau$	function type
		$\mid$	$\tau \times \tau$	product type
		$\mid$	$\top$	unit type
Terms	t	$::=$	x	term variables
		$\mid$	$f\ t$	term constructors
		$\mid$	(t, t)	products
		$\mid$	$()$	unit
		$\mid$	X	predicate variables
		$\mid$	p	predicate names
Atoms	A	$::=$	$p\ t$	atoms
		$\mid$	$X\ t$	parametric atoms
LHS	l	$::=$	$\cdot$	empty left-hand side
		$\mid$	A, l	left-hand side extension
Rules	R	$::=$	$l \multimap P$	rewrite
		$\mid$	$\forall x : \iota.\, R$	term abstraction
		$\mid$	$\forall X : \tau \to o.\, R$	predicate abstraction
Programs	P	$::=$	$\cdot$	empty program
		$\mid$	P, P	program union
		$\mid$	A	atom
		$\mid$	$!A$	reusable atom
		$\mid$	R	rule
		$\mid$	$!R$	reusable rule
		$\mid$	$\exists x : \tau \to \iota.\, P$	new term variable
		$\mid$	$\exists X : \tau \to o.\, P$	new predicate variable

Figure A.1: Grammar of $\mathcal{L}^{1.5}$

A.2 Typing

This appendix summarizes the typing rules of our core language, without regard for safety constraint checks (which are the subject of Appendix A.3) or the module syntax (which is given in Appendix A.5).

Typing makes use of signatures and contexts, defined next.

$$
\begin{array}{rrcll}
\textit{Signatures} & \Sigma & ::= & \cdot & \text{empty signature} \\
 & & \mid & \Sigma, f : \tau \to \iota & \text{constructor declaration} \\
 & & \mid & \Sigma, p : \tau \to o & \text{predicate declaration} \\
\textit{Contexts} & \Gamma & ::= & \cdot & \text{empty context} \\
 & & \mid & \Gamma, x : \iota & \text{term assumption} \\
 & & \mid & \Gamma, X : \tau \to o & \text{predicate assumption}
\end{array}
$$

The typing rules for our language are very conventional. They are as follows.

Terms $\boxed{\Gamma \vdash_\Sigma t : \tau}$

$$\frac{}{\Gamma, x : \tau \vdash_\Sigma x : \tau} \qquad \frac{\Gamma \vdash_{\Sigma, f:\tau\to\iota} t : \tau}{\Gamma \vdash_{\Sigma, f:\tau\to\iota} f\ t : \iota} \qquad \frac{\Gamma \vdash_\Sigma t_1 : \tau_1 \quad \Gamma \vdash_\Sigma t_2 : \tau_2}{\Gamma \vdash_\Sigma (t_1, t_2) : \tau_1 \times \tau_2} \qquad \frac{}{\Gamma \vdash_\Sigma () : \top}$$

$$\frac{}{\Gamma, X : \tau \to o \vdash_\Sigma X : \tau \to o} \qquad \frac{}{\Gamma \vdash_{\Sigma, p:\tau\to o} p : \tau \to o}$$

Atoms $\boxed{\Gamma \vdash_\Sigma A\ atom}$

$$\frac{\Gamma \vdash_{\Sigma, p:\tau\to o} t : \tau}{\Gamma \vdash_{\Sigma, p:\tau\to o} p\ t\ atom} \qquad \frac{\Gamma, X : \tau \to o \vdash_\Sigma t : \tau}{\Gamma, X : \tau \to o \vdash_\Sigma X\ t\ atom}$$

Rule left-hand-sides $\boxed{\Gamma \vdash_\Sigma l\ lhs}$

$$\frac{}{\Gamma \vdash_\Sigma \cdot\ lhs} \qquad \frac{\Gamma \vdash_\Sigma A\ atom \quad \Gamma \vdash_\Sigma l\ lhs}{\Gamma \vdash_\Sigma A, l\ lhs}$$

Rules $\boxed{\Gamma \vdash_\Sigma R\ rule}$

$$\frac{\Gamma \vdash_\Sigma l\ lhs \quad \Gamma \vdash_\Sigma P\ prog}{\Gamma \vdash_\Sigma l \multimap P\ rule} \qquad \frac{\Gamma, x : \iota \vdash_\Sigma R\ rule}{\Gamma \vdash_\Sigma \forall x : \iota.\, R\ rule} \qquad \frac{\Gamma, X : \tau \to o \vdash_\Sigma R\ rule}{\Gamma \vdash_\Sigma \forall X : \tau \to o.\, R\ rule}$$

Programs $\boxed{\Gamma \vdash_\Sigma P\ prog}$

$$\frac{}{\Gamma \vdash_\Sigma \cdot\ prog} \qquad \frac{\Gamma \vdash_\Sigma P_1\ prog \quad \Gamma \vdash_\Sigma P_2\ prog}{\Gamma \vdash_\Sigma P_1, P_2\ prog}$$

$$\frac{\Gamma \vdash_\Sigma A\ atom}{\Gamma \vdash_\Sigma A\ prog} \qquad \frac{\Gamma \vdash_\Sigma A\ atom}{\Gamma \vdash_\Sigma !A\ prog} \qquad \frac{\Gamma \vdash_\Sigma R\ rule}{\Gamma \vdash_\Sigma R\ prog} \qquad \frac{\Gamma \vdash_\Sigma R\ rule}{\Gamma \vdash_\Sigma !R\ prog}$$

$$\frac{\Gamma \vdash_{\Sigma, x:\tau\to\iota} P\ prog}{\Gamma \vdash_\Sigma \exists x : \tau \to \iota.\, P\ prog} \qquad \frac{\Gamma \vdash_{\Sigma, X:\tau\to o} P\ prog}{\Gamma \vdash_\Sigma \exists X : \tau \to o.\, P\ prog}$$

Although evaluation level entities, we also define archives and states and give their typing rules.

$$
\begin{array}{rrcll}
\textit{Archives} & \Omega & ::= & \cdot & \text{empty archive} \\
 & & | & \Omega, A & \text{reusable atom} \\
 & & | & \Omega, R & \text{reusable rule} \\
\textit{States} & \Sigma.\langle \Omega \ ; \ \Pi \rangle & & &
\end{array}
$$

The typing rules for states amount to requiring that the contained archive and program be valid and closed.

Archives $\boxed{\vdash_\Sigma \Omega \ \textit{archive}}$

$$\frac{}{\vdash_\Sigma \cdot \ \textit{archive}} \qquad \frac{\vdash_\Sigma \Omega \ \textit{archive} \quad \cdot \vdash_\Sigma A \ \textit{atom}}{\vdash_\Sigma \Omega, A \ \textit{archive}} \qquad \frac{\vdash_\Sigma \Omega \ \textit{archive} \quad \cdot \vdash_\Sigma R \ \textit{rule}}{\vdash_\Sigma \Omega, R \ \textit{archive}}$$

States $\boxed{\vdash \Sigma.\langle \Omega \ ; \ \Pi \rangle \ \textit{state}}$

$$\frac{\vdash_\Sigma \Omega \ \textit{archive} \quad \cdot \vdash_\Sigma \Pi \ \textit{prog}}{\vdash \Sigma.\langle \Omega \ ; \ \Pi \rangle \ \textit{state}}$$

A.3 Typing with Safety Checks

In this appendix, we refine the typing rules seen in Appendix A.2 to capture the safety constraints on variable occurrences in a program.

An atom A in a program is a *left-hand side atom* (or lhs-atom) if it is introduced by the production $l ::= A, l$ in the grammar of Appendix A.1. It is a *right-hand side atom* (a rhs-atom) if it is instead introduced by the productions $P ::= A$ and $P ::= !A$. We write A_L and A_R for a generic lhs- and rhs-atom, respectively. Terms occurring in a lhs-atom (rhs-atom) are called lhs-terms (rhs-terms, respectively). We write $A_L[x]$ to indicate that the variable x occurs in a term in lhs-atom A_L, and similarly for rhs-atoms. We adopt the same notation for predicate variables.

The safety constraints can then be expressed as follows:

1. If a (term or predicate) variable introduced by universal quantification occurs in a rhs-atom, this same variable must also occur in an earlier lhs-atom.

   Pictorially, every occurrence of a rhs-atom $A_R[x]$ within the scope of a universal declaration for x shall adhere to the following template

   $$\forall x.\ \cdots A_L[x] \cdots \multimap (\cdots, A_R[x], \cdots) \cdots$$

   The case for a predicate variable X is analogous.

2. If a universal predicate variable X leads a parametric lhs-atom $X\ t$, then X must occur in a term position either in the same left-hand side or in a the left-hand side of an enclosing rule. If the first case, it must be possible to rearrange this left-hand side so that the term occurrence appears to the left of the $X\ t$ to avoid circularities such as $X\ (Y), Y\ (X)$.

   Pictorially, given any program, it should be possible to reorder its lhs-atoms so that, for every lhs-atom $X\ t$ where X is universal, there is another lhs-atom $A_L[X]$ so that either $A_L[X]$ occurs in the left-hand side of an outer rule, as in the following template,

   $$\forall X.\ \cdots A_L[X] \cdots \multimap (\cdots, X\ t, \cdots \multimap \cdots)$$

   or $A_L[X]$ occurs to the left of $X\ t$ in the same left-hand side, as depicted by the following template:

   $$\forall X.\ \cdots (\cdots, A_L[X], \cdots, X\ t, \cdots) \multimap \cdots$$

We enforce the safety constraints on variables by augmenting the typing judgments in Appendix A.2 with either one or two *constraint sets*. A constraint set annotates each universal variable in scope with a marker that indicates whether this variable has been seen in a lhs-term before. Specifically, x^0 indicates that variable x has been declared (i.e., introduced by means of a universal quantifier) but has not yet appeared in a term in a lhs-atom. Instead, x^1 designates x as having been seen in an earlier lhs-atom. The same convention applies for predicate variables.

Constraint sets are therefore defined by the following grammar:

Constraints	$\mathcal{X}$	$::=$	$\cdot$	no constraints
		$\mid$	$\mathcal{X}, x^0$	unseen term variable
		$\mid$	$\mathcal{X}, x^1$	seen term variable
		$\mid$	$\mathcal{X}, X^0$	unseen predicate variable
		$\mid$	$\mathcal{X}, X^1$	seen predicate variable

The typing judgments seen in Appendix A.2 acquire a new component, a constraint set that tracks the use of universal variables. The judgments for syntactic objects that appear in a program position (including rule right-hand sides) are updated differently from the judgments for objects that appear in a left-hand side. Since atoms and terms can appear in both, they are refined into two versions, one for when they occur in program position (rhs-atoms and rhs-terms) and a distinct one for when they appear in a left-hand side (lhs-atoms and lhs-terms).

Program-side judgments are extended with a single constraint set, which we indicate in red. These judgments are

$\Gamma \vdash_\Sigma t : \tau \mid \mathcal{X}$	Terms (rhs)
$\Gamma \vdash_\Sigma A\ atom \mid \mathcal{X}$	Atoms (rhs)
$\Gamma \vdash_\Sigma R\ rule \mid \mathcal{X}$	Rules
$\Gamma \vdash_\Sigma P\ prog \mid \mathcal{X}$	Programs

In each rule, the constraint set $\mathcal{X}$ is extended as the corresponding syntactic object is traversed, going from the conclusion to the premises of the rules. The markers of the variables in $\mathcal{X}$ are checked in rules where variables are used (in the judgments for terms and atoms).

Left-hand side judgments are enriched by two constraint sets, which we write as $\mathcal{X} > \mathcal{X}'$ and color in blue. These judgments are

$\Gamma \vdash_\Sigma t : \tau \mid \mathcal{X} > \mathcal{X}'$	Terms (lhs)
$\Gamma \vdash_\Sigma A\ atom \mid \mathcal{X} > \mathcal{X}'$	Atoms (lhs)
$\Gamma \vdash_\Sigma l\ lhs \mid \mathcal{X} > X'$	Left-hand sides

The constraint set $\mathcal{X}$ is understood as an input to the judgment while $\mathcal{X}'$ is its output: $\mathcal{X}$ is propagated from the conclusion to one of the premises of a rule while $\mathcal{X}'$ flows the other direction, from a premise to the conclusion. The set $\mathcal{X}'$ differs from $\mathcal{X}$ by the fact that the marker of some variables has been updated from ^{0} to ^{1}. This takes place when a variable is encountered within a (left-hand side) term

The updated rules are displayed next, with commentary when needed to clarify the refinement. Here, we take advantage of the commutativity of "," and implicitly reorder atoms in a left-hand side as needed to obtain a derivation. It is not necessary to do so in in program positions. In all cases, we treat existential variables as if they were constants.

Terms (lhs) $\boxed{\Gamma \vdash_\Sigma t : \tau \mid \mathcal{X} > \mathcal{X}'}$

$$\frac{}{\Gamma, x : \tau \vdash_\Sigma x : \tau \mid \mathcal{X}, x^? > \mathcal{X}, x^1} \qquad \frac{\Gamma \vdash_{\Sigma, f:\tau\to\iota} t : \tau \mid \mathcal{X} > \mathcal{X}'}{\Gamma \vdash_{\Sigma, f:\tau\to\iota} f\ t : \iota \mid \mathcal{X} > \mathcal{X}'}$$

$$\frac{\Gamma \vdash_\Sigma t_1 : \tau_1 \mid \mathcal{X} > \mathcal{X}' \quad \Gamma \vdash_\Sigma t_2 : \tau_2 \mid \mathcal{X}' > \mathcal{X}''}{\Gamma \vdash_\Sigma (t_1, t_2) : \tau_1 \times \tau_2 \mid \mathcal{X} > \mathcal{X}''} \qquad \frac{}{\Gamma \vdash_\Sigma () : \top \mid \mathcal{X} > \mathcal{X}}$$

$$\frac{}{\Gamma, X : \tau \to o \vdash_\Sigma X : \tau \to o \mid \mathcal{X}, X^? > \mathcal{X}, X^1} \qquad \frac{}{\Gamma \vdash_{\Sigma, p:\tau\to o} p : \tau \to o \mid \mathcal{X} > \mathcal{X}}$$

The rules for term and predicate variables update the marker for that variable to ^{1}, whatever it was before — we write $x^?$ for either x^0 or x^1 and similarly for predicate variables. The other rules thread the variables sets through. We arbitrarily proceed from left to right in the case of pairs.

Terms (rhs) $\boxed{\Gamma \vdash_\Sigma t : \tau \mid \mathcal{X}}$

$$\frac{}{\Gamma, x : \tau \vdash_\Sigma x : \tau \mid \mathcal{X}, x^1} \qquad \frac{\Gamma \vdash_{\Sigma, f:\tau\to\iota} t : \tau \mid \mathcal{X}}{\Gamma \vdash_{\Sigma, f:\tau\to\iota} f\ t : \iota \mid \mathcal{X}} \qquad \frac{\Gamma \vdash_\Sigma t_1 : \tau_1 \mid \mathcal{X} \quad \Gamma \vdash_\Sigma t_2 : \tau_2 \mid \mathcal{X}}{\Gamma \vdash_\Sigma (t_1, t_2) : \tau_1 \times \tau_2 \mid \mathcal{X}} \qquad \frac{}{\Gamma \vdash_\Sigma () : \top \mid \mathcal{X}}$$

$$\frac{}{\Gamma, X : \tau \to o \vdash_\Sigma X : \tau \to o \mid \mathcal{X}, X^1} \qquad \frac{}{\Gamma \vdash_{\Sigma, p:\tau\to o} p : \tau \to o \mid \mathcal{X}}$$

The right-hand side rules for term and predicate variables require that these variables have been seen earlier in a left-hand side, which is witnessed by the marker ^{1}. All other rules thread the constraint set through.

Atoms (lhs) $\boxed{\Gamma \vdash_\Sigma A\ atom \mid \mathcal{X} > \mathcal{X}'}$

$$\frac{\Gamma \vdash_{\Sigma, p:\tau\to o} t : \tau \mid \mathcal{X} > \mathcal{X}'}{\Gamma \vdash_{\Sigma, p:\tau\to o} p\ t\ atom \mid \mathcal{X} > \mathcal{X}'} \qquad \frac{\Gamma, X : \tau \to o \vdash_\Sigma t : \tau \mid \mathcal{X}, X^1 > \mathcal{X}'}{\Gamma, X : \tau \to o \vdash_\Sigma X\ t\ atom \mid \mathcal{X}, X^1 > \mathcal{X}'}$$

A predicate variable X introducing a parametric atom on the left-hand side must have been seen earlier within a left-hand side term, as indicated by the marker X^1. Other variables are threaded through.

Atoms (rhs) $\boxed{\Gamma \vdash_\Sigma A\ atom \mid \mathcal{X}}$

$$\frac{\Gamma \vdash_{\Sigma, p:\tau\to o} t : \tau \mid \mathcal{X}}{\Gamma \vdash_{\Sigma, p:\tau\to o} p\ t\ atom \mid \mathcal{X}} \qquad \frac{\Gamma, X : \tau \to o \vdash_\Sigma t : \tau \mid \mathcal{X}, X^1}{\Gamma, X : \tau \to o \vdash_\Sigma X\ t\ atom \mid \mathcal{X}, X^1}$$

Like any right-hand side variable, a predicate variable introducing a parametric atom must have been seen previously within a left-hand side term.

Rule left-hand-sides $\boxed{\Gamma \vdash_\Sigma l\ lhs \mid \mathcal{X} > \mathcal{X}'}$

$$\frac{}{\Gamma \vdash_\Sigma \cdot\ lhs \mid \mathcal{X} > \mathcal{X}} \qquad \frac{\Gamma \vdash_\Sigma A\ atom \mid \mathcal{X} > \mathcal{X}' \quad \Gamma \vdash_\Sigma l\ lhs \mid \mathcal{X}' > \mathcal{X}''}{\Gamma \vdash_\Sigma A, l\ lhs \mid \mathcal{X} > \mathcal{X}''}$$

Constraint variables are refined (i.e., some markers ^{0} are upgraded to ^{1}) going from left to right in a left-hand side.

Rules $\boxed{\Gamma \vdash_\Sigma R\ rule \mid \mathcal{X}}$

$$\frac{\Gamma \vdash_\Sigma l\ lhs \mid \mathcal{X} > \mathcal{X}' \quad \Gamma \vdash_\Sigma P\ prog \mid \mathcal{X}'}{\Gamma \vdash_\Sigma l \multimap P\ rule \mid \mathcal{X}} \qquad \frac{\Gamma, x : \iota \vdash_\Sigma R\ rule \mid \mathcal{X}, x^0}{\Gamma \vdash_\Sigma \forall x : \iota.\, R\ rule \mid \mathcal{X}} \qquad \frac{\Gamma, X : \tau \to o \vdash_\Sigma R\ rule \mid \mathcal{X}, X^0}{\Gamma \vdash_\Sigma \forall X : \tau \to o.\, R\ rule \mid \mathcal{X}}$$

The typing judgment for rules is where a lot of the action takes place. Traversing a universal quantifier installs its variable in the constraint set in the premise with marker ^{0}. A bare rule of the form $l \multimap P$ is handled by passing the current constraint set $\mathcal{X}$ to the left-hand side l, which will refine it into the constraint set $\mathcal{X}'$ which is what is passed to the program part P.

Programs $\boxed{\Gamma \vdash_\Sigma P\ prog \mid \mathcal{X}}$

$$\frac{}{\Gamma \vdash_\Sigma \cdot\ prog \mid \mathcal{X}} \qquad \frac{\Gamma \vdash_\Sigma P_1\ prog \mid \mathcal{X} \quad \Gamma \vdash_\Sigma P_2\ prog \mid \mathcal{X}}{\Gamma \vdash_\Sigma P_1, P_2\ prog \mid \mathcal{X}} \qquad \frac{\Gamma \vdash_\Sigma A\ atom \mid \mathcal{X}}{\Gamma \vdash_\Sigma A\ prog \mid \mathcal{X}} \qquad \frac{\Gamma \vdash_\Sigma A\ atom \mid \mathcal{X}}{\Gamma \vdash_\Sigma !A\ prog \mid \mathcal{X}}$$

$$\frac{\Gamma \vdash_\Sigma R\ rule \mid \mathcal{X}}{\Gamma \vdash_\Sigma R\ prog \mid \mathcal{X}} \qquad \frac{\Gamma \vdash_\Sigma R\ rule \mid \mathcal{X}}{\Gamma \vdash_\Sigma !R\ prog \mid \mathcal{X}} \qquad \frac{\Gamma \vdash_{\Sigma, x:\tau\to\iota} P\ prog \mid \mathcal{X}}{\Gamma \vdash_\Sigma \exists x : \tau \to \iota.\, P\ prog \mid \mathcal{X}} \qquad \frac{\Gamma \vdash_{\Sigma, X:\tau\to o} P\ prog \mid \mathcal{X}}{\Gamma \vdash_\Sigma \exists X : \tau \to o.\, P\ prog \mid \mathcal{X}}$$

The typing rules for programs simply pass the constraint set in their conclusion to their premise. Note in particular that existential variables are ignored.

The typing judgment for archives and states remains unchanged, but the rules need to be upgraded to seed their constituents with an empty constraint set.

Archives $\boxed{\vdash_\Sigma \Omega\ archive}$

$$\frac{}{\vdash_\Sigma \cdot\ archive} \qquad \frac{\vdash_\Sigma \Omega\ archive \quad \cdot \vdash_\Sigma A\ atom \mid \cdot}{\vdash_\Sigma \Omega, A\ archive} \qquad \frac{\vdash_\Sigma \Omega\ archive \quad \cdot \vdash_\Sigma R\ rule \mid \cdot}{\vdash_\Sigma \Omega, R\ archive}$$

States $\boxed{\vdash \Sigma.\langle \Omega\ ;\ \Pi\rangle\ state}$

$$\frac{\vdash_\Sigma \Omega\ archive \quad \cdot \vdash_\Sigma \Pi\ prog \mid \cdot}{\vdash \Sigma.\langle \Omega\ ;\ \Pi\rangle\ state}$$

A.4 First-Order Encoding

In this appendix, we formalize the encoding that transforms an $\mathcal{L}^{1.5}$ state into a state in $\mathcal{L}^{1}$ with the same typing and operational behavior. To do so, we need to extend this encoding to all entities of $\mathcal{L}^{1.5}$. Intuitively, we will map every universal second-order variable $X : \tau \to o$ to a first-order variables $x_X : \iota$ and replace each parametric atom $X\ t$ with the atom $p_\tau(x_X, t)$, where p_τ is a predicate name associated with type τ. Occurrences of X in a term will be translated to just x_X. Existential second-order variables X are treated slightly differently because existential first-order variables must have a type of the form $\tau \to \iota$. Therefore, we will associate to each X a first-order variable $x_X : \top \to \iota$ and proceed as in the universal case, except that wherever we had x_X we now have $x_X\ ()$.

The encoding below assumes that predicates occurring within terms are not themselves applied to terms, i.e., that they have the bare form p rather than $p\ t$. This will simplify the presentation. We can always rewrite a program that makes use of applied predicates by turning the application into a pair. For example, the term $f\ (p\ t)$ would be rewritten as $f\ (p, t)$. This transformation can be done automatically. A consequence of this simplification is that the type o cannot appear immediately underneath a product or as the antecedent of a function type, but only in the form $\tau \to o$.

As we encounter second-order variables in terms and atoms, we need to know whether they are universal or existential. We do so by maintaining a set $\mathcal{X}$ which marks each variable declaration in scope as universal or existential. Its structure is defined by means of the following grammar:

$$\begin{array}{llcll} \textit{Declaration sets} & \mathcal{X} & ::= & \cdot & \text{empty set} \\ & & \mid & \mathcal{X}, \forall X : \tau \to o & \text{universal declaration} \\ & & \mid & \mathcal{X}, \exists X : \tau \to o & \text{existential declaration} \end{array}$$

We now describe the encoding $[O]^\sharp_{\mathcal{X}}$ of each syntactic entity O in $\mathcal{L}^{1.5}$, providing commentary as needed.

Terms $\boxed{[t]^\sharp_{\mathcal{X}} = t'}$

$$\begin{cases} [x]^\sharp_{\mathcal{X}} &= x \\ [f\ t]^\sharp_{\mathcal{X}} &= f\ [t]^\sharp_{\mathcal{X}} \\ [(t_1, t_2)]^\sharp_{\mathcal{X}} &= ([t_1]^\sharp_{\mathcal{X}}, [t_2]^\sharp_{\mathcal{X}}) \\ [()]^\sharp_{\mathcal{X}} &= () \\ [X\ t]^\sharp_{\mathcal{X}, \forall X:\tau\to o} &= x_X \\ [X\ t]^\sharp_{\mathcal{X}, \exists X:\tau\to o} &= x_X\ () \\ [p]^\sharp_{\mathcal{X}} &= x_p\ () \end{cases}$$

Terms are translated homomorphically except for second-order variables and predicate names. A variable X is mapped to x_X if it is universal and to $x_X\ ()$ if it is existential. Predicate names can occur in $\mathcal{L}^{1.5}$ terms and are treated as if they were introduced by an existential quantifier. In fact, we will later associate a function symbol $x_p : \top \to \iota$ to each predicate name p in the signature.

Atoms $\boxed{[A]^\sharp_{\mathcal{X}} = A'}$

$$\begin{cases} [p\ t]^\sharp_{\mathcal{X}} &= p\ [t]^\sharp_{\mathcal{X}} \\ [X\ t]^\sharp_{\mathcal{X}, \forall X:\tau\to o} &= p_\tau\ (x_X, [t]^\sharp_{\mathcal{X}, \forall X:\tau\to o}) \\ [X\ t]^\sharp_{\mathcal{X}, \exists X:\tau\to o} &= p_\tau\ (x_X\ (), [t]^\sharp_{\mathcal{X}, \exists X:\tau\to o}) \end{cases}$$

A second-order variable $X : \tau \to o$ heading a parametric predicate $X\ t$ is replaced by the predicate p_τ associated with the type τ — we will see how this is done momentarily. We remember X by passing either x_X or $x_X\ ()$ as the first argument of p_τ.

Rule left-hand sides $\boxed{[l]^\sharp_{\mathcal{X}} = l'}$

$$\begin{cases} [\cdot]^\sharp_{\mathcal{X}} &= \cdot \\ [A, l]^\sharp_{\mathcal{X}} &= [A]^\sharp_{\mathcal{X}}, [l]^\sharp_{\mathcal{X}} \end{cases}$$

Left-hand sides simply propagates the translation to their constituent atoms.

Rules $\boxed{[R]^\sharp_{\mathcal{X}} = \langle \Sigma'; R' \rangle}$

$$\begin{cases} [l \multimap P]^\sharp_{\mathcal{X}} &= \langle \Sigma'; [l]^\sharp_{\mathcal{X}} \multimap P' \rangle & \text{where } [P]^\sharp_{\mathcal{X}} = \langle \Sigma'; P' \rangle \\ [\forall x : \iota.\, R]^\sharp_{\mathcal{X}} &= \langle \Sigma'; \forall x : \iota.\, R' \rangle & \text{where } [R]^\sharp_{\mathcal{X}} = \langle \Sigma'; R' \rangle \\ [\forall X : \tau \to o.\, R]^\sharp_{\mathcal{X}} &= \langle (\Sigma' \cup p_\tau : \iota \times \tau' \to o); \forall x_X : \iota.\, R' \rangle & \text{where } [R]^\sharp_{\mathcal{X}, \forall X : \tau \to o} = \langle \Sigma'; R' \rangle \end{cases}$$

Translating a rule diverges from the above pattern because we may need to account for new second-order variables. The interesting case is that of second-order universal rules $\forall X : \tau \to o.\, R$. The matrix R may make use of X, which means that while translating it we need to know that X was introduced universally. Therefore, we extend the declaration set with a universal marker $\forall X : \tau \to o$ while doing so. Now, if X heads a parametric predicate $X\ t$ inside R, it will get replaced by the predicate p_τ associated to τ. We remember which such predicates are needed by returning a signature Σ' as an additional output of the translation function. Because the translation of t may replace some predicate names and variables with terms, the type of p_τ is $\iota \times \tau' \to o$ where τ' is the translation of the type τ, defined later. Note that the predicate name p_τ is fixed. Therefore, were the translation of R to turn out another copy, we shall retain just one. We write $\Sigma \cup \Sigma'$ for the set union of signatures Σ and Σ'. It keep just one copy of any common declaration. This differs from the notation "$\Sigma, p_\tau : \iota \times \tau' \to o$" which α-renames p_τ apart from Σ.

Programs $\boxed{[P]^\sharp_{\mathcal{X}} = \langle \Sigma'; P' \rangle}$

$$\begin{cases} [\cdot]^\sharp_{\mathcal{X}} &= \langle \cdot; \cdot \rangle & \\ [P_1, P_2]^\sharp_{\mathcal{X}} &= \langle \Sigma'_1 \cup \Sigma'_2; (P'_1, P'_2) \rangle & \text{where } [P_1]^\sharp_{\mathcal{X}} = \langle \Sigma'_1; P'_1 \rangle \text{ and } [P_2]^\sharp_{\mathcal{X}} = \langle \Sigma'_2; P'_2 \rangle \\ [A]^\sharp_{\mathcal{X}} &= \langle \cdot; [A]^\sharp_{\mathcal{X}} \rangle & \\ [!A]^\sharp_{\mathcal{X}} &= \langle \cdot; ![A]^\sharp_{\mathcal{X}} \rangle & \\ [R]^\sharp_{\mathcal{X}} &= [R]^\sharp_{\mathcal{X}} & \\ [!R]^\sharp_{\mathcal{X}} &= \langle \Sigma'; !R' \rangle & \text{where } [R]^\sharp_{\mathcal{X}} = \langle \Sigma'; R' \rangle \\ [\exists x : \tau \to \iota.\, P]^\sharp_{\mathcal{X}} &= \langle \Sigma'; \exists x : \tau^\sharp \to \iota.\, P' \rangle & \text{where } [P]^\sharp_{\mathcal{X}} = \langle \Sigma'; P' \rangle \\ [\exists X : \tau \to o.\, P]^\sharp_{\mathcal{X}} &= \langle (\Sigma' \cup p_\tau : \iota \times \tau^\sharp \to o); \exists x_X : \top \to \iota.\, P' \rangle & \text{where } [P]^\sharp_{\mathcal{X}, \exists X : \tau \to o} = \langle \Sigma'; P' \rangle \end{cases}$$

The translation of programs follows the template we just saw for rules. This time, the variables are existential.

We now give a translation for types, which is used in the encoding of rules and programs as we saw. We also give translations for signatures, contexts and archives.

Types $\boxed{\tau^\sharp = \tau'}$

$$\begin{cases} \iota^\sharp &= \iota & \\ o^\sharp &= \iota & \textit{(unused)} \\ (\tau_1 \to \iota)^\sharp &= {\tau'_1}^\sharp \to \iota & \\ (\tau_1 \to o)^\sharp &= \top \to \iota & \\ (\tau_1 \to \tau_2)^\sharp &= \iota & \textit{(unused for } \tau \neq \iota \textit{ and } \tau \neq o\textit{)} \\ (\tau_1 \times \tau_2)^\sharp &= {\tau_1}^\sharp \times {\tau_2}^\sharp & \\ \top^\sharp &= \top & \end{cases}$$

The type τ translated by $\tau^\sharp$ will always occur on the left-hand side of function type $\tau \to \tau'$. In fact τ' can be only ι or o in $\mathcal{L}^{1.5}$. Because we restricted predicate name occurrences within terms to be bare, the type o can only appear as the target of a function type, of the form $\tau' \to o$. In particular isolated occurrences of o will never be encountered. We arbitrarily map both unused forms to ι.

Signatures $\boxed{[\Sigma]^\sharp = \Sigma'}$

$$\begin{cases} [\cdot]^\sharp & = & \cdot \\ [\Sigma, f : \tau \to \iota]^\sharp & = & [\Sigma]^\sharp, f : \tau^\sharp \to \iota \\ [\Sigma, p : \tau \to o]^\sharp & = & [\Sigma]^\sharp, p : \tau^\sharp \to o, x_p : \top \to \iota \end{cases}$$

The translation of signatures bears much in common with what we did for existential variables. Specifically, we associate to each predicate name $p : \tau \to o$ not only its natural encoding $p : \tau^\sharp \to o$ but also the function symbol $x_p : \top \to \iota$ in case this predicate name is used in a term (see our encoding of terms above).

Contexts $\boxed{[\Gamma]^\sharp = \langle \Sigma'; \Gamma' \rangle}$

$$\begin{cases} [\cdot]^\sharp & = & \langle \cdot; \cdot \rangle & \\ [\Gamma, x : \iota]^\sharp & = & \langle \Sigma'; \Gamma', x : \iota \rangle & \text{where } [\Gamma]^\sharp = \langle \Sigma'; \Gamma' \rangle \\ [\Gamma, X : \tau \to o]^\sharp & = & \langle (\Sigma' \cup p_\tau : \iota \times \tau^\sharp \to o); \Gamma', x_X : \iota \rangle & \text{where } [\Gamma]^\sharp = \langle \Sigma'; \Gamma' \rangle \end{cases}$$

We encode a predicate variable $X : \tau \to o$ in a context into a first-order variable x_X of type ι, but also define the signature declaration for p_τ, which will account for parametric atoms $X\ t$ that may make use of X. Therefore, the translation of a context returns both a signature and another context.

Archives $\boxed{[\Omega]^\sharp = \langle \Sigma'; \Omega' \rangle}$

$$\begin{cases} [\cdot]^\sharp & = & \langle \cdot; \cdot \rangle & \\ [\Omega, A]^\sharp & = & \langle \Sigma'; (\Omega', A^\sharp) \rangle & \text{where } [\Omega]^\sharp = \langle \Sigma'; \Omega' \rangle \\ [\Omega, R]^\sharp & = & \langle \Sigma' \cup \Sigma''; (\Omega', R') \rangle & \text{where } [\Omega]^\sharp = \langle \Sigma'; \Omega' \rangle \text{ and } [R]^\sharp_\cdot = \langle \Sigma''; R' \rangle \end{cases}$$

A archive is encoded by collecting the encoding of each constituent. Since the encoding of rules output a signature, so does the encoding of archives.

The above translation maps a well-typed entity in $\mathcal{L}^{1.5}$ to a similarly well-typed object in $\mathcal{L}^1$. This is formally expressed by the following property.

Lemma 5 *For $\mathcal{L}^{1.5}$ signature Σ and context Γ, let $[\Sigma]^\sharp = \Sigma'$ and $[\Gamma]^\sharp = \langle \Sigma''; \Gamma' \rangle$.*

1. *If $\Gamma \vdash_\Sigma t : \tau$ in $\mathcal{L}^{1.5}$, then $\Gamma' \vdash_{\Sigma' \cup \Sigma''} [t : \tau^\sharp]^\sharp_\cdot$ in $\mathcal{L}^1$.*
2. *If $\Gamma \vdash_\Sigma A$ atom in $\mathcal{L}^{1.5}$, then $\Gamma' \vdash_{\Sigma' \cup \Sigma''} [A]^\sharp_\cdot$ atom in $\mathcal{L}^1$.*
3. *If $\Gamma \vdash_\Sigma l$ lhs in $\mathcal{L}^{1.5}$, then $\Gamma' \vdash_{\Sigma' \cup \Sigma''} [l]^\sharp_\cdot$ lhs in $\mathcal{L}^1$.*
4. *If $\Gamma \vdash_\Sigma R$ rule in $\mathcal{L}^{1.5}$, then $\Gamma' \vdash_{\Sigma' \cup \Sigma'' \cup \Sigma'''} R'$ rule in $\mathcal{L}^1$ where $[R]^\sharp_\cdot = \langle \Sigma'''; R' \rangle$.*
5. *If $\Gamma \vdash_\Sigma P$ prog in $\mathcal{L}^{1.5}$, then $\Gamma' \vdash_{\Sigma' \cup \Sigma'' \cup \Sigma'''} P'$ prog in $\mathcal{L}^1$ where $[P]^\sharp_\cdot = \langle \Sigma'''; P' \rangle$.*
6. *If $\vdash_\Sigma \Omega$ archive in $\mathcal{L}^{1.5}$, then $\vdash_{\Sigma' \cup \Sigma'''} \Omega'$ archive in $\mathcal{L}^1$ where $[\Omega]^\sharp = \langle \Sigma'''; \Omega' \rangle$.*

7. *If* $\vdash \Sigma.\langle \Omega \ ; \ \Pi \rangle$ *state in* $\mathcal{L}^{1.5}$, *then* $\vdash (\Sigma' \cup \Sigma''' \cup \Sigma'''').\langle \Omega' \ ; \ \Pi' \rangle$ *state in* $\mathcal{L}^{1}$ *where* $[\Omega]^\sharp = \langle \Sigma'''; \Omega' \rangle$ *and* $[\Pi]^\sharp_\cdot = \langle \Sigma''''; \Pi' \rangle$.

Proof The proof proceeds by mutual induction on the given typing derivations. □

The encoding also maps transitions in $\mathcal{L}^{1.5}$ to transitions in $\mathcal{L}^{1}$.

Lemma 6 *For* $\mathcal{L}^{1.5}$ *state* $\Sigma_A.\langle \Omega_A \ ; \ \Pi_A \rangle$, *let* $[\Sigma_A]^\sharp = \Sigma'_A$ *and* $[\Omega_A]^\sharp = \langle \Sigma''_A; \Omega'_A \rangle$ *and* $[\Pi_A]^\sharp_\cdot = \langle \Sigma'''_A; \Pi'_A \rangle$.
Similarly, for $\mathcal{L}^{1.5}$ *state* $\Sigma_B.\langle \Omega_B \ ; \ \Pi_B \rangle$, *let* $[\Sigma_B]^\sharp = \Sigma'_B$ *and* $[\Omega_B]^\sharp = \langle \Sigma''_B; \Omega'_B \rangle$ *and* $[\Pi_B]^\sharp_\cdot = \langle \Sigma'''_B; \Pi'_B \rangle$.
Let $\Sigma^*_A = \Sigma'_A \cup \Sigma''_A \cup \Sigma'''_A$ *and* $\Sigma^*_B = \Sigma'_B \cup \Sigma''_B \cup \Sigma'''_B$.
If $\vdash \Sigma_A.\langle \Omega_A \ ; \ \Pi_A \rangle$ *state and* $\Sigma_A.\langle \Omega_A \ ; \ \Pi_A \rangle \mapsto \Sigma_B.\langle \Omega_B \ ; \ \Pi_B \rangle$ *in* $\mathcal{L}^{1.5}$, *then* $\Sigma^*_A.\langle \Omega'_A \ ; \ \Pi'_A \rangle \mapsto \Sigma^*_B.\langle \Omega'_B \ ; \ \Pi'_B \rangle$ *in* $\mathcal{L}^{1}$.

Proof The proof proceeds by cases on the given $\mathcal{L}^{1.5}$ transitions. □

A.5 Typechecking Modular Programs

This appendix defines the extension $\mathcal{L}^M$ of the language $\mathcal{L}^{1.5}$ with the module infrastructure devised in Section 3. We describe both the syntax of $\mathcal{L}^M$ and its extended typing semantics, inclusive of mode declarations and checks. Extensions are shown in blue. Differently from Section 4 and Appendix A.4, we do not compile $\mathcal{L}^M$ into $\mathcal{L}^{1.5}$ but handle it head on. Furthermore, we allow the programmer to mix and match the module syntax that characterizes $\mathcal{L}^M$ with more basic constructs of $\mathcal{L}^{1.5}$. For example, she will be able to define a module using the **module** syntax but use it without relying on **as**, and vice versa. We conclude by summarizing the elaboration of $\mathcal{L}^M$ into $\mathcal{L}^{1.5}$.

The basic syntactic categories of $\mathcal{L}^M$ extend those of $\mathcal{L}^{1.5}$ with module reference names N. The overall lexicon is as follows, where we make the same provisos as for $\mathcal{L}^{1.5}$ in Appendix A.1.

Term variables	x
Predicate variables	X
Function symbols	f
Predicate symbols	p
Module references	N

The other entities of $\mathcal{L}^M$ are defined by the grammar in Figure A.2, where all modifications (not just extensions) are highlighted in blue. These modifications are as follows:

- The type of state objects has been refined into the form o^{ξ} to incorporate mode ascriptions for predicate names, as used in the **provide** stanza of the examples in Section 3. The type o^{l} refers to the predicate names whose atoms can only appear in the left-hand side of a rule (except at their definition point) — that's the marker **in**, o^{r} stands for predicate names that can occur only on the right-hand side (again, except when being defined) — that's the marker **out**, and o indicates predicate names that can occur anywhere — they correspond to the type o of $\mathcal{L}^{1.5}$. We allow mode restrictions on any predicate name, not just the symbols exported by a module.

  Given an atom type o^{ξ}, we write $\overline{o^{\xi}}$ for the type that flips the restrictions on use. Specifically,

$$\begin{cases} \overline{o^{\mathrm{l}}} & = & o^{\mathrm{r}} \\ \overline{o^{\mathrm{r}}} & = & o^{\mathrm{l}} \\ \overline{o} & = & o \end{cases}$$

  This is useful as module definitions provide predicates to be used on the left-hand side in client code by producing them on their right-hand side, and vice versa.

- A predicate variable V is either one of the predicate variable X of $\mathcal{L}^{1.5}$ or a compound predicate variable $N.X$ where N is a module reference. Either form can be existentially or universally quantified. Similarly, existential term variables (and later term-level signature declaration) can be either simple or compound names.

- An interface Υ is a multiset of declarations for function and predicate symbols. It is what is exported by a module. It also corresponds exactly to the signatures of $\mathcal{L}^{1.5}$. Although they did not appear in any of the examples in Section 3, term-level symbols can be exported by a module. As we will see, interfaces in $\mathcal{L}^M$ are richer.

- Programs are extended with the construct for using a module and the construct for defining one.

The resulting grammar is given in Figure A.2.

We proceed by giving typing rules for $\mathcal{L}^M$, as just defined. We update the definition of signature to allow compound term and predicate names and the definition of context to allow moded, possibly compound, predicate declarations.

Compound term variables cannot enter a context.

Signatures	Σ	$::=$	$\cdot$	empty signature
		$\mid$	$\Sigma, f : \tau \to \iota$	constructor declaration
		$\mid$	$\Sigma, N.f : \tau \to \iota$	compound constructor declaration
		$\mid$	$\Sigma, p : \tau \to o^{\xi}$	predicate declaration
		$\mid$	$\Sigma, N.p : \tau \to o^{\xi}$	compound predicate declaration
Contexts	Γ	$::=$	$\cdot$	empty context
		$\mid$	$\Gamma, x : \iota$	term assumption
		$\mid$	$\Gamma, V : \tau \to o^{\xi}$	predicate assumption

The definition of archives and states remains unchanged.

Two extensions to the typing semantics of $\mathcal{L}^{1.5}$ are required. First we need to check that predicate variables are used according to their mode declaration. This entails providing different rules for atoms occurring on the left- and right-hand side of a rule. Second, we need to keep track of the modules available for instantiation. To do so, we define *module sets* M, which associate the name and interface of each module in scope.

Module set	M	$::=$	$\cdot$	empty module set
		$\mid$	$M, p : \Upsilon$	available module

The judgments for terms and states remain unchanged. We update the judgments for atoms, rules and programs as follows:

$\Gamma \vdash_{\Sigma} t : \tau \ \mathit{lhs\ atom}$	Atoms (left-hand side)
$\Gamma \vdash_{\Sigma} A \ \mathit{rhs\ atom}$	Atoms (right-hand side)
$\Gamma \vdash_{\Sigma} R \ \mathit{rule} \mid M$	Rules
$\Gamma \vdash_{\Sigma} P \ \mathit{prog} \mid M > M'$	Programs

When analyzing a program P, the set M represents known modules before examining P while M' is the extended module set after having gone through P. Scope considerations prevent a rule from exporting more modules than it knows about.

We now give the updated typing rules, highlighting the changes in blue. We comment on changes as required.

Terms $\boxed{\Gamma \vdash_{\Sigma} t : \tau}$

$$\frac{}{\Gamma, x : \tau \vdash_{\Sigma} x : \tau} \qquad \frac{\Gamma \vdash_{\Sigma, f:\tau\to\iota} t : \tau}{\Gamma \vdash_{\Sigma, f:\tau\to\iota} f\ t : \iota} \qquad \frac{\Gamma \vdash_{\Sigma, N.f:\tau\to\iota} t : \tau}{\Gamma \vdash_{\Sigma, N.f:\tau\to\iota} N.f\ t : \iota}$$

$$\frac{\Gamma \vdash_{\Sigma} t_1 : \tau_1 \quad \Gamma \vdash_{\Sigma} t_2 : \tau_2}{\Gamma \vdash_{\Sigma} (t_1, t_2) : \tau_1 \times \tau_2} \qquad \frac{}{\Gamma \vdash_{\Sigma} () : \top} \qquad \frac{}{\Gamma, V : \tau \to o^{\xi} \vdash_{\Sigma} V : \tau \to o^{\xi}} \qquad \frac{}{\Gamma \vdash_{\Sigma, p:\tau\to o^{\xi}} p : \tau \to o^{\xi}}$$

The rules for terms change only in as much as predicate variables may use compound names and carry modes, and that we could have compound term constructors in the signature.

Atoms (lhs) $\boxed{\Gamma \vdash_{\Sigma} A \ \mathit{lhs\ atom}}$

$$\frac{\Gamma \vdash_{\Sigma, p:\tau\to o^{\mathrm{l?}}} t : \tau}{\Gamma \vdash_{\Sigma, p:\tau\to o^{\mathrm{l?}}} p\ t \ \mathit{lhs\ atom}} \qquad \frac{\Gamma, V : \tau \to o^{\mathrm{l?}} \vdash_{\Sigma} t : \tau}{\Gamma, V : \tau \to o^{\mathrm{l?}} \vdash_{\Sigma} V\ t \ \mathit{lhs\ atom}}$$

Here, we write $o^{\mathrm{l?}}$ for either type o or o^{l}. Predicate names in a left-hand side atom can be either unrestricted (o) or defined for left-hand side use only (o^{l}).

Atoms (rhs) $\boxed{\Gamma \vdash_\Sigma A \ rhs \ atom}$

$$\frac{\Gamma \vdash_{\Sigma, p:\tau \to o^{\mathbf{r}?}} t : \tau}{\Gamma \vdash_{\Sigma, p:\tau \to o^{\mathbf{r}?}} p\ t \ rhs \ atom} \qquad \frac{\Gamma, V : \tau \to o^{\mathbf{r}?} \vdash_\Sigma t : \tau}{\Gamma, V : \tau \to o^{\mathbf{r}?} \vdash_\Sigma V\ t \ rhs \ atom}$$

Similar considerations apply to right-hand side atoms. Here, $o^{\mathbf{r}?}$ stands for either o or $o^{\mathbf{r}}$.

Rule left-hand-sides $\boxed{\Gamma \vdash_\Sigma l \ lhs}$

$$\frac{}{\Gamma \vdash_\Sigma \cdot \ lhs} \qquad \frac{\Gamma \vdash_\Sigma A \ lhs \ atom \quad \Gamma \vdash_\Sigma l \ lhs}{\Gamma \vdash_\Sigma A, l \ lhs}$$

Rule left-hand sides are updated only in so far as their constituents are sent to the left-hand side atom judgment.

Rules $\boxed{\Gamma \vdash_\Sigma R \ rule \mid M}$

$$\frac{\Gamma \vdash_\Sigma l \ lhs \quad \Gamma \vdash_\Sigma P \ prog \mid M > M'}{\Gamma \vdash_\Sigma l \multimap P \ rule \mid M} \qquad \frac{\Gamma, x : \iota \vdash_\Sigma R \ rule \mid M}{\Gamma \vdash_\Sigma \forall x : \iota . R \ rule \mid M} \qquad \frac{\Gamma, V : \tau \to o^\xi \vdash_\Sigma R \ rule \mid M}{\Gamma \vdash_\Sigma \forall V : \tau \to o^\xi . R \ rule \mid M}$$

Rules are where we first encounter module sets. All they do is to propagate them to the program on the right-hand side P of a rewriting directive $l \multimap P$. Note that the module set M' that comes back is discarded as the modules defined in P are not in scope outside of P itself.

Programs $\boxed{\Gamma \vdash_\Sigma P \ prog \mid M > M'}$

$$\frac{}{\Gamma \vdash_\Sigma \cdot \ prog \mid M > M} \qquad \frac{\Gamma \vdash_\Sigma P_1 \ prog \mid M > M' \quad \Gamma \vdash_\Sigma P_2 \ prog \mid M' > M''}{\Gamma \vdash_\Sigma P_1, P_2 \ prog \mid M > M''}$$

$$\frac{\Gamma \vdash_\Sigma A \ rhs \ atom}{\Gamma \vdash_\Sigma A \ prog \mid M > M} \qquad \frac{\Gamma \vdash_\Sigma A \ rhs \ atom}{\Gamma \vdash_\Sigma !A \ prog \mid M > M} \qquad \frac{\Gamma \vdash_\Sigma R \ rule \mid M}{\Gamma \vdash_\Sigma R \ prog \mid M > M} \qquad \frac{\Gamma \vdash_\Sigma R \ rule \mid M}{\Gamma \vdash_\Sigma !R \ prog \mid M > M}$$

$$\frac{\Gamma \vdash_{\Sigma, v:\tau \to \iota} P \ prog \mid M > M'}{\Gamma \vdash_\Sigma \exists v : \tau \to \iota . P \ prog \mid M > M'} \qquad \frac{\Gamma \vdash_{\Sigma, V:\tau \to o^\xi} P \ prog \mid M > M'}{\Gamma \vdash_\Sigma \exists V : \tau \to o^\xi . P \ prog \mid M > M'}$$

$$\frac{\Gamma \vdash_\Sigma p\ t \ lhs \ atom \quad \Gamma \vdash_{\Sigma, N.\Upsilon} P \ prog \mid M, p : \Upsilon > M'}{\Gamma \vdash_\Sigma N \ \mathbf{as} \ p\ t . P \ prog \mid M, p : \Upsilon > M'}$$

$$\frac{p : \Upsilon_{par} \times \Upsilon_{export} \to o^{\mathbf{r}} \text{ in } \Sigma \qquad p \text{ not in } M \qquad \Gamma, \Upsilon_{par}, \overline{\Upsilon_{export}} \vdash_{\Sigma, \Upsilon_{local}} P \ prog \mid M, p : \Upsilon_{export} > M'}{\Gamma \vdash_\Sigma \mathbf{module}\ p(\Upsilon_{par})\ \mathbf{provide}\ \Upsilon_{export}\ \mathbf{local}\ \Upsilon_{local}\ P\ \mathbf{end}\ prog \mid M > M'}$$

The typing rules for programs are where most of the action takes place. Notice first that the union P_1, P_2 of two programs accumulates modules from left to right. As we understand "," as being commutative and associative, this does not force a module to be defined before being used in a strictly lexical sense. Because some ordering of a program constituents must be picked, it however prevents circularity. Naturally, the empty program and right-hand side atoms simply return the module set they were handed. The updated version of the remaining rules of $\mathcal{L}^{1.5}$ simply propagate the sets to and from their premises.

The rules for the new constructs are more interesting. The rule for a module definition

$$\mathbf{module}\ p(\Upsilon_{par})\ \mathbf{provide}\ \Upsilon_{export}\ \mathbf{local}\ \Upsilon_{local}\ P\ \mathbf{end}$$

first checks that the predicate name p (the name of the module) is defined in the signature Σ and that it takes as input both the parameters Υ_{par} and the module's interface Υ_{export}. Abusing notation, we write $\Upsilon_{par} \times \Upsilon_{export}$ for the overall product type — for notational simplicity we keep the parameters on the left and the exported symbols on the right although there is no reason for this. Since a hand-crafted use of this module (i.e., without relying on **as**) would make use of it on the right-hand side of a rule, we give p mode $o^{\mathbf{r}}$ as using it on the left-hand side of a rule (other than the above module definition) would have the effect of intercepting the instantiation request. Therefore, we require that p appears in the signature Σ with type $\Upsilon_{par} \times \Upsilon_{export} \to o^{\mathbf{r}}$. The second premise ensures that module p is defined no more than once.

The last premise of this rule extends the context with both the parameter declarations Υ_{par} and the inverted exported declarations Υ_{export}, and the signature with the local declarations Υ_{local}. We invert the modes of the exported declarations as the module uses these symbols in the opposite way as a client. The module set that can be used inside P is extended with $p : \Upsilon_{export}$, thereby allowing recursive instantiations. Note that this extension will be returned through M'.

The rule for the instantiation N **as** $p\,t.\,P$ of a module p with interface Υ, as reported in the incoming module set, starts by checking the validity of $p\,t$ as a right-hand side atom. It then extends the signature in which to typecheck P with the exported symbols Υ prefixed by the module reference name N. This is expressed by the notation $N.\Upsilon$.

Archives $\boxed{\vdash_\Sigma \Omega\ \mathit{archive} \mid M}$

$$\frac{}{\vdash_\Sigma \cdot\ \mathit{archive} \mid M} \qquad \frac{\vdash_\Sigma \Omega\ \mathit{archive} \mid M \quad \cdot \vdash_\Sigma A\ \mathit{rhs\ atom}}{\vdash_\Sigma \Omega, A\ \mathit{archive} \mid M} \qquad \frac{\vdash_\Sigma \Omega\ \mathit{archive} \mid M \quad \cdot \vdash_\Sigma R\ \mathit{rule} \mid M}{\vdash_\Sigma \Omega, R\ \mathit{archive} \mid M}$$

The rules for archives pass the module set they are given to any contained $\mathcal{L}^M$ rule. As we will see next when examining states, this module set will be empty in this report.

States $\boxed{\vdash \Sigma.\langle \Omega\ ;\ \Pi\rangle\ \mathit{state}}$

$$\frac{\vdash_\Sigma \Omega\ \mathit{archive} \mid \cdot \quad \cdot \vdash_\Sigma \Pi\ \mathit{prog} \mid \cdot > M'}{\vdash \Sigma.\langle \Omega\ ;\ \Pi\rangle\ \mathit{state}}$$

The rule for states seeds the embedded archive Ω and program Π with an empty module set and discards what the analysis of Π reports. Note that, were an implementation to provide system modules, this is where they would be plugged in.

A program in $\mathcal{L}^M$ is elaborated into an $\mathcal{L}^{1.5}$ program before execution. This means that we can rely on the operational semantics for this language, as discussed in Section 2.1.3, for running $\mathcal{L}^M$ programs, without a need to account for any of the modular infrastructure. The elaboration, adapted from Section 4, is defined as as follows:

- A module declaration

  $$\begin{array}{l}\mathbf{module}\ p(\Upsilon_{par})\\ \quad \mathbf{provide}\quad \Upsilon_{export}\\ \quad \mathbf{local}\quad \Upsilon_{local}\\ \quad P\\ \mathbf{end}\end{array}$$

  is compiled into the rule

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

  where the predicate name p is called on the names declared in Υ_{par} and Υ_{export}, dropping the types, something we express as $p(\Upsilon^*_{par}, \Upsilon^*_{export})$.

- Let p be a module with interface Υ, which we write as $\vec{X} : \vec{\tau}$ for convenience — note that $\vec{X}$ may contain term variables as well as predicate variables. Then, a module instantiation

$$N \ \mathbf{as}\ p\, t.\, P$$

is compiled into the program

$$\exists \Upsilon.\ \left[\begin{array}{l} p\,(t, \vec{X}), \\ [\vec{X}/N.\vec{X}]/P \end{array}\right]$$

where $[\vec{X}/N.\vec{X}]/P$ replaces each occurrence of the compound name $N.X_i$ in P with X_i, and similarly for term variables. We implicitly rename variables as required to avoid capture.

Starting with a well-typed $\mathcal{L}^M$ program without compound predicate names in its signature, this transformation results in a valid $\mathcal{L}^{1.5}$ program without compound predicate names at all. Mode annotations are dropped altogether.

Atom types	o^{ξ}	::=	o	unrestricted atom
		|	$o^{\mathbf{l}}$	lhs-only atom
		|	$o^{\mathbf{r}}$	rhs-only atom
Types	τ	::=	ι	individuals
		|	o^{ξ}	state objects
		|	$\tau \to \tau$	function type
		|	$\tau \times \tau$	product type
		|	$\top$	unit type
Interface	Υ	::=	$\cdot$	empty interface
		|	$\Upsilon, f : \tau \to \iota$	function symbol
		|	$\Upsilon, p : \tau \to o^{\xi}$	predicate symbol
Term variables	v	::=	x	simple term variable
		|	$N.x$	compound term variable
Predicate variables	V	::=	X	simple predicate variable
		|	$N.X$	compound predicate variable
Terms	t	::=	v	term variables
		|	$f\ t$	term constructors
		|	(t, t)	products
		|	$()$	unit
		|	V	predicate variables
		|	p	predicate names
Atoms	A	::=	$p\ t$	atoms
		|	$V\ t$	parametric atoms
LHS	l	::=	$\cdot$	empty left-hand side
		|	A, l	left-hand side extension
Rules	R	::=	$l \multimap P$	rewrite
		|	$\forall x : \iota.\, R$	term abstraction
		|	$\forall V : \tau \to o^{\xi}.\, R$	predicate abstraction
Programs	P	::=	$\cdot$	empty program
		|	P, P	program union
		|	A	atom
		|	$!A$	reusable atom
		|	R	rule
		|	$!R$	reusable rule
		|	$\exists v : \tau \to \iota.\, P$	new term variable
		|	$\exists V : \tau \to o^{\xi}.\, P$	new predicate variable
		|	N **as** $p\ t.\, P$	module instantiation
		|	**module** $p(\Upsilon)$	module definition
			provide Υ	
			local Υ	
			P	
			end	

Figure A.2: Syntax of $\mathcal{L}^M$

A.6 Congruence

Throughout this report, we implicitly viewed the operator "," used to combine left-hand side atoms and (right-hand side) programs as associative with unit "$\cdot$" and commutative. This is logically justified as the corresponding operators in linear logic, $\otimes$ and $\mathbf{1}$, have isomorphic properties relative to derivability — see Appendix B.

For the sake of completeness, we spell out these properties in the case of left-hand sides:

$$\begin{array}{rcll} l_1, (l_2, l_3) & = & (l_1, l_2), l_3 & \text{associativity} \\ l, \cdot & = & l & \text{unit} \\ l_1, l_2 & = & l_2, l_1 & \text{commutativity} \end{array}$$

Analogous properties apply to programs. These properties form an equivalence relation. Moreover, they apply arbitrarily deep within a term and therefore define a congruence.

We also wrote "," for the operation of extending (and joining) signatures, contexts and state components and "$\cdot$" for the empty signature, context, archive and state. Also in these case, "," is commutative and associative with unit "$\cdot$" — this corresponds directly to the often-used conventions for the homonymous formation operators in logic.

The use of "," as a pairing operator in terms is neither associative nor commutative.

A.7 Unfocused Rewriting Semantics

The full set of transitions in the unfocused evaluation semantics is as follows.

Transitions $\boxed{\Sigma.\langle \Omega \ ; \ \Pi\rangle \mapsto \Sigma'.\langle \Omega' \ ; \ \Pi'\rangle}$

$$
\begin{array}{rcll}
\Sigma.\langle \Omega, l_1 \ ; \ \Pi, l_2, (l_1, l_2) \multimap P\rangle & \mapsto & \Sigma.\langle \Omega, l_1 \ ; \ \Pi, P\rangle & \\
\Sigma.\langle \Omega \ ; \ \Pi, \forall x : \iota.\, R\rangle & \mapsto & \Sigma.\langle \Omega \ ; \ \Pi, [t/x]R\rangle & \text{if } \cdot \vdash_\Sigma t : \iota \\
\Sigma.\langle \Omega \ ; \ \Pi, \forall X : \tau \to o.\, R\rangle & \mapsto & \Sigma.\langle \Omega \ ; \ \Pi, [p/X]R\rangle & \text{if } p : \tau \to o \text{ in } \Sigma \\
\Sigma.\langle \Omega \ ; \ \Pi, !A\rangle & \mapsto & \Sigma.\langle \Omega, A \ ; \ \Pi\rangle & \\
\Sigma.\langle \Omega \ ; \ \Pi, !R\rangle & \mapsto & \Sigma.\langle \Omega, R \ ; \ \Pi\rangle & \\
\Sigma.\langle \Omega \ ; \ \Pi, \exists x : \tau \to \iota.\, P\rangle & \mapsto & (\Sigma, x : \tau \to \iota).\langle \Omega \ ; \ \Pi, P\rangle & \\
\Sigma.\langle \Omega \ ; \ \Pi, \exists X : \tau \to o.\, P\rangle & \mapsto & (\Sigma, X : \tau \to o).\langle \Omega \ ; \ \Pi, P\rangle & \\
\Sigma.\langle \Omega, R \ ; \ \Pi\rangle & \mapsto & \Sigma.\langle \Omega, R \ ; \ \Pi, R\rangle &
\end{array}
$$

The first three transitions refer to the rules of $\mathcal{L}^{1.5}$ according to the definition in Appendix A.1. The next four handles the remaining program forms. The last rule transfers a copy of an archived rule to the program part of the state for further processing. Archived atoms are used in the first rule, which processes rewriting directives. As we will see, each transition corresponds to the left sequent rule of the corresponding logical operator, as given in Appendix B.1. Observe that we are implicitly making use of the congruences in Appendix A.6 in the way we organize the starting state of each transition.

With the partial exception of ! which we decomposed over several rules, these transitions view each operator in the language as a local state transformation directive. We can make this correspondence tighter by doing away with the archive component of a state and merging the rules that process ! with the rules that use the resulting program components. The following presentation does precisely this:

Transitions $\boxed{\Sigma.\,\langle \Pi\rangle \mapsto \Sigma'.\,\langle \Pi'\rangle}$

$$
\begin{array}{rcll}
\Sigma.\,\langle \Pi, !(l_1), l_2, (l_1, l_2) \multimap P\rangle & \mapsto & \Sigma.\,\langle \Pi, !(l_1), P\rangle & \\
\Sigma.\,\langle \Pi, \forall x : \iota.\, R\rangle & \mapsto & \Sigma.\,\langle \Pi, [t/x]R\rangle & \text{if } \cdot \vdash_\Sigma t : \iota \\
\Sigma.\,\langle \Pi, \forall X : \tau \to o.\, R\rangle & \mapsto & \Sigma.\,\langle \Pi, [p/X]R\rangle & \text{if } p : \tau \to o \text{ in } \Sigma \\
\Sigma.\,\langle \Pi, !R\rangle & \mapsto & \Sigma.\,\langle \Pi, !R, R\rangle & \\
\Sigma.\,\langle \Pi, \exists x : \tau \to \iota.\, P\rangle & \mapsto & (\Sigma, x : \tau \to \iota).\,\langle \Pi, P\rangle & \\
\Sigma.\,\langle \Pi, \exists X : \tau \to o.\, P\rangle & \mapsto & (\Sigma, X : \tau \to o).\,\langle \Pi, P\rangle &
\end{array}
$$

where $!(l)$ denotes a portion of the state that consists of zero or more reusable atoms. This presentation is equivalent to the transitions at the beginning of this section and it too has its roots in linear logic. It is however somewhat harder to work with. We will not pursue this presentation further.

A.8 Focused Rewriting Semantics

The focused operational semantics relies on the notion of a stable state $\Sigma.\langle \Omega \ ; \ \Pi \rangle$, whose program component Π is defined as follows:

$$\begin{array}{rrcll} \textit{Stable state program} & \Pi & ::= & \cdot & \text{empty state} \\ & & | & \Pi, A & \text{atom} \\ & & | & \Pi, R & \text{rule} \end{array}$$

The full set of focused transitions is as given next.

Rule transitions $\boxed{\Sigma.\langle \Omega \ ; \ \Pi \rangle \Mapsto \Sigma'.\langle \Omega' \ ; \ \Pi' \rangle}$

$$\begin{array}{rcl} \Sigma.\langle \Omega, l_1\theta \ ; \ \Pi, l_2\theta, \forall(l_1, l_2 \multimap P) \rangle & \Mapsto & \Sigma.\langle \Omega, l_1\theta \ ; \ \Pi, P\theta \rangle \\ \Sigma.\langle \Omega, l_1\theta, \forall(l_1, l_2 \multimap P) \ ; \ \Pi, l_2\theta \rangle & \Mapsto & \Sigma.\langle \Omega, l_1\theta, \forall(l_1, l_2 \multimap P) \ ; \ \Pi, P\theta \rangle \end{array}$$

Stabilizing transitions $\boxed{\Sigma.\langle \Omega \ ; \ \Pi \rangle \Mapsto \Sigma'.\langle \Omega \ ; \ \Pi' \rangle}$

$$\begin{array}{rcl} \Sigma.\langle \Omega \ ; \ \Pi, !A \rangle & \Mapsto & \Sigma.\langle \Omega, A \ ; \ \Pi \rangle \\ \Sigma.\langle \Omega \ ; \ \Pi, !R \rangle & \Mapsto & \Sigma.\langle \Omega, R \ ; \ \Pi \rangle \\ \Sigma.\langle \Omega \ ; \ \Pi, \exists x : \tau \to \iota.\, P \rangle & \Mapsto & (\Sigma, x : \tau \to \iota).\langle \Omega \ ; \ \Pi, P \rangle \\ \Sigma.\langle \Omega \ ; \ \Pi, \exists X : \tau \to o.\, P \rangle & \Mapsto & (\Sigma, X : \tau \to o).\langle \Omega \ ; \ \Pi, P \rangle \end{array}$$

As we will see in Appendix B.2, these transitions stem from the interpretation of $\mathcal{L}^{1.5}$ in focused intuitionistic logic. As for the unfocused rules, we have implicitly used the congruences in Appendix A.6 in this presentation.

To facilitate the parallel with focused linear logic, it is useful to give a second presentation of the focused operational semantics of $\mathcal{L}^{1.5}$, and specifically of the rule transition judgment $\Sigma.\langle \Omega \ ; \ \Pi \rangle \Mapsto \Sigma'.\langle \Omega \ ; \ \Pi' \rangle$. We introduce an additional rule processing judgment $\Sigma.\langle \Omega \ ; \ \Pi, \boxed{R} \rangle \mapsto \Sigma'.\langle \Omega \ ; \ \Pi' \rangle$ and define both by inference rules as follows:

Rule selection $\boxed{\Sigma.\langle \Omega \ ; \ \Pi \rangle \Mapsto \Sigma'.\langle \Omega' \ ; \ \Pi' \rangle}$

$$\frac{\Sigma.\langle \Omega \ ; \ \Pi, \boxed{R} \rangle \Mapsto \Sigma'.\langle \Omega' \ ; \ \Pi' \rangle}{\Sigma.\langle \Omega \ ; \ \Pi, R \rangle \Mapsto \Sigma'.\langle \Omega' \ ; \ \Pi' \rangle} \qquad \frac{\Sigma.\langle \Omega \ ; \ \Pi, \boxed{R} \rangle \Mapsto \Sigma'.\langle \Omega' \ ; \ \Pi' \rangle}{\Sigma.\langle \Omega, R \ ; \ \Pi \rangle \Mapsto \Sigma'.\langle \Omega' \ ; \ \Pi' \rangle}$$

Rule processing $\boxed{\Sigma.\langle \Omega \ ; \ \Pi, \boxed{R} \rangle \Mapsto \Sigma'.\langle \Omega' \ ; \ \Pi' \rangle}$

$$\frac{}{\Sigma.\langle \Omega, l_1 \ ; \ \Pi, l_2, \boxed{l_1, l_2 \multimap P} \rangle \Mapsto \Sigma.\langle \Omega, l_1 \ ; \ \Pi, P \rangle}$$

$$\frac{\cdot \vdash_\Sigma t : \iota \quad \Sigma.\langle \Omega \ ; \ \Pi, \boxed{[t/x]R} \rangle \Mapsto \Sigma'.\langle \Omega' \ ; \ \Pi' \rangle}{\Sigma.\langle \Omega \ ; \ \Pi, \boxed{\forall x : \iota.\, R} \rangle \Mapsto \Sigma'.\langle \Omega' \ ; \ \Pi' \rangle} \qquad \frac{(\Sigma, p : \tau \to o).\langle \Omega \ ; \ \Pi, \boxed{[p/X]R} \rangle \Mapsto \Sigma'.\langle \Omega' \ ; \ \Pi' \rangle}{(\Sigma, p : \tau \to o).\langle \Omega \ ; \ \Pi, \boxed{\forall X : \tau \to o.\, R} \rangle \Mapsto \Sigma'.\langle \Omega \ ; \ \Pi' \rangle}$$

It is easy to show that the transition $\Sigma.\langle \Omega \ ; \ \Pi \rangle \Mapsto \Sigma'.\langle \Omega' \ ; \ \Pi' \rangle$ at the beginning of this section is achievable if and only if there is a derivation of $\Sigma.\langle \Omega \ ; \ \Pi \rangle \Mapsto \Sigma'.\langle \Omega' \ ; \ \Pi' \rangle$ according to the above rules. The rule selection rules matches closely the focus rules of focused linear logic, while the rule processing rules correspond to the chaining of appropriate non-invertible rules. This is discussed in further detail in Appendix B.2.

We can go one step further and directly define the judgment $\Sigma.\langle \Omega \ ; \ \Pi \rangle \Rightarrow \Sigma'.\langle \Omega' \ ; \ \Pi' \rangle$ mentioned in Section 2.1.3. Recall that this judgment maps stable states to stable states by selecting one rule to apply and then stabilizing the resulting state. It is realized by the following rules:

Stable transition $\boxed{\Sigma.\langle \Omega \ ; \ \Pi \rangle \Rrightarrow \Sigma'.\langle \Omega' \ ; \ \Pi' \rangle}$

$$\frac{\Sigma.\langle \Omega \ ; \ \Pi, \boxed{R} \rangle \Mapsto \Sigma'.\langle \Omega' \ ; \ \Pi' \rangle}{\Sigma.\langle \Omega \ ; \ \Pi, R \rangle \Rrightarrow \Sigma'.\langle \Omega' \ ; \ \Pi' \rangle} \qquad \frac{\Sigma.\langle \Omega, R \ ; \ \Pi, \boxed{R} \rangle \Mapsto \Sigma'.\langle \Omega' \ ; \ \Pi' \rangle}{\Sigma.\langle \Omega, R \ ; \ \Pi \rangle \Rrightarrow S'.\langle \Omega' \ ; \ \Pi' \rangle}$$

Rule processing $\boxed{\Sigma.\langle \Omega \ ; \ \Pi, \boxed{R} \rangle \Mapsto \Sigma'.\langle \Omega' \ ; \ \Pi' \rangle}$

$$\frac{\Sigma.\langle \Omega, l_1 \ ; \ \Pi, P \rangle \Mapsto \Sigma'.\langle \Omega' \ ; \ \Pi' \rangle}{\Sigma.\langle \Omega, l_1 \ ; \ \Pi, l_2, \boxed{l_1, l_2 \multimap P} \rangle \Mapsto \Sigma'.\langle \Omega' \ ; \ \Pi' \rangle}$$

$$\frac{\cdot \vdash_\Sigma t : \iota \quad \Sigma.\langle \Omega \ ; \ \Pi, \boxed{[t/x]R} \rangle \Mapsto \Sigma'.\langle \Omega' \ ; \ \Pi' \rangle}{\Sigma.\langle \Omega \ ; \ \Pi, \boxed{\forall x : \iota.\, R} \rangle \Mapsto \Sigma'.\langle \Omega' \ ; \ \Pi' \rangle} \qquad \frac{(\Sigma, p : \tau \to o).\langle \Omega \ ; \ \Pi, \boxed{[p/X]R} \rangle \Mapsto \Sigma'.\langle \Omega' \ ; \ \Pi' \rangle}{(\Sigma, p : \tau \to o).\langle \Omega \ ; \ \Pi, \boxed{\forall X : \tau \to o.\, R} \rangle \Mapsto \Sigma'.\langle \Omega' \ ; \ \Pi' \rangle}$$

Stabilization $\boxed{\Sigma.\langle \Omega \ ; \ \Pi \rangle \Mapsto \Sigma'.\langle \Omega' \ ; \ \Pi' \rangle}$

$$\frac{}{\Sigma.\langle \Omega \ ; \ \Pi \rangle \Mapsto \Sigma.\langle \Omega \ ; \ \Pi \rangle}$$

$$\frac{\Sigma.\langle \Omega, A \ ; \ \Pi \rangle \Mapsto \Sigma'.\langle \Omega' \ ; \ \Pi' \rangle}{\Sigma.\langle \Omega \ ; \ \Pi, !A \rangle \Mapsto \Sigma'.\langle \Omega' \ ; \ \Pi' \rangle} \qquad \frac{\Sigma.\langle \Omega, R \ ; \ \Pi \rangle \Mapsto \Sigma'.\langle \Omega' \ ; \ \Pi' \rangle}{\Sigma.\langle \Omega \ ; \ \Pi, !R \rangle \Mapsto \Sigma'.\langle \Omega' \ ; \ \Pi' \rangle}$$

$$\frac{(\Sigma, x : \tau \to \iota).\langle \Omega \ ; \ \Pi, P \rangle \Mapsto \Sigma'.\langle \Omega' \ ; \ \Pi' \rangle}{\Sigma.\langle \Omega \ ; \ \Pi, \exists x : \tau \to \iota.\, P \rangle \Mapsto \Sigma'.\langle \Omega' \ ; \ \Pi' \rangle} \qquad \frac{(\Sigma, X : \tau \to o).\langle \Omega \ ; \ \Pi, P \rangle \Mapsto \Sigma'.\langle \Omega' \ ; \ \Pi' \rangle}{\Sigma.\langle \Omega \ ; \ \Pi, \exists X : \tau \to o.\, P \rangle \Mapsto \Sigma'.\langle \Omega' \ ; \ \Pi' \rangle}$$

The last four rules corresponds to invertible rules in focused linear logic.

We could simplify the operational semantics of $\mathcal{L}^{1.5}$ even further by observing that the existentials in a program P can always be brought outward in prenex position (possibly after renaming bound variables). Therefore, every program P is equivalent to a program of the form $\exists \Sigma.\, P_\Pi, !(P_\Omega)$, where P_Π is a collection of atoms A and rules R, and $!(P_\Omega)$ denotes a similar collection of reusable atoms and rules. Notice that, when closed, P_Π has the form of a stable state program and P_Ω of an archive. This all implies that a rule R matches the template $\forall (l \multimap \exists \Sigma.\, P_\Pi, !(P_\Omega))$.

Then, the above rules collapse into the following two rules, each of which carries out a single step of execution that transforms a stable state into another stable state.

$$\frac{}{\Sigma.\langle \Omega, l_1\theta \ ; \ \Pi, l_2\theta, \forall (l_1, l_2 \multimap \exists \Sigma'.\, P_\Pi, !(P_\Omega)) \rangle \Rrightarrow (\Sigma, \Sigma').\langle \Omega, l_1\theta, P_\Omega\theta \ ; \ \Pi, P_\Pi\theta \rangle}$$

$$\frac{}{\Sigma.\langle \underbrace{\Omega, l_1\theta, \forall (l_1, l_2 \multimap \exists \Sigma'.\, P_\Pi, !(P_\Omega))}_{\Omega^*} \ ; \ \Pi, l_2\theta \rangle \Rrightarrow (\Sigma, \Sigma').\langle \Omega^*, P_\Omega\theta \ ; \ \Pi, P_\Pi\theta \rangle}$$

We can rewrite them as state transitions over stable states as follows:

$$\begin{array}{lcl} \Sigma.\langle \Omega, l_1\theta \ ; \ \Pi, l_2\theta, \forall (l_1, l_2 \multimap \exists \Sigma'.\, P_\Pi, !(P_\Omega)) \rangle & \Rrightarrow & (\Sigma, \Sigma').\langle \Omega, l_1\theta, P_\Omega\theta \ ; \ \Pi, P_\Pi\theta \rangle \\ \Sigma.\langle \underbrace{\Omega, l_1\theta, \forall (l_1, l_2 \multimap \exists \Sigma'.\, P_\Pi, !(P_\Omega))}_{\Omega^*} \ ; \ \Pi, l_2\theta \rangle & \Rrightarrow & (\Sigma, \Sigma').\langle \Omega^*, P_\Omega\theta \ ; \ \Pi, P_\Pi\theta \rangle \end{array}$$

Although possible in this language, this last simplification may not be easily expressible in a larger language, for example were we to include the choice operator $\&$ of [9].

B Logical Interpretation

This appendix discusses a fragment of intuitionistic linear logic that is related to the language examined in this report. Section B.1 defines this fragment, multiplicative-exponential intuitionistic linear logic or MEILL, and shows its traditional derivation rules. Section B.2 gives a focused presentation. Section B.3 identifies the precise fragment of MEILL our core language corresponds to and relates its rewriting semantics both to traditional and focused derivability. For the sake of brevity, we do not distinguish between first- and second-order variables or quantification at the logical level.

B.1 Multiplicative-exponential Intuitionistic Linear Logic

Multiplicative-exponential intuitionistic linear logic, MEILL for short, is the fragment of intuitionistic linear logic [17] deprived of additive connectives and units.

Its formulas, denoted φ and ψ possibly with subscripts and superscripts, are defined by the following grammar:

$$\textit{Formulas} \quad \varphi \quad ::= \quad A \mid \mathbf{1} \mid \varphi \otimes \varphi \mid \varphi \multimap \varphi \mid !\varphi \mid \forall x : \tau . \varphi \mid \exists x : \tau . \varphi$$

Atomic formulas A are constructed as in $\mathcal{L}^{1,5}$. In particular, symbols wherein are drawn from a signature Σ.

We describe derivability for MEILL in the sequent calculus style using a two-context judgment of the form

$$\Gamma ; \Delta \longrightarrow_{\Sigma} \varphi$$

where Γ and Δ are multisets of formulas called the persistent and linear context respectively. A context is defined as follows:

$$\begin{array}{rrcll} \textit{Contexts} & \Delta & ::= & \cdot & \text{Empty context} \\ & & \mid & \Delta, \varphi & \text{Context extension} \end{array}$$

We will often use "," as a context union operator and consider it commutative and associative with unit "$\cdot$". In a sequent $\Gamma ; \Delta \longrightarrow_{\Sigma} \varphi$, the linear context Δ holds formulas that can be used exactly once in a derivation of φ, while the formulas in the persistent context Γ can be used arbitrarily many times.

The rules defining the above judgment are given in Figure B.1. The cut rules for this language are admissible [31] and have therefore been omitted. We have grayed out rules for the fragment of MEILL that do not correspond to any operator in our rewriting language (see Appendix B.3 for details).

Observe that, aside for $\otimes$ and $\mathbf{1}$ which play a structural role, only left rules are retained.

$$\frac{}{\Gamma; \varphi \longrightarrow_\Sigma \varphi}\ \mathsf{init} \qquad \frac{\Gamma, \varphi; \Delta, \varphi \longrightarrow_\Sigma \psi}{\Gamma, \varphi; \Delta \longrightarrow_\Sigma \psi}\ \mathsf{clone}$$

$$\frac{\Gamma; \Delta \longrightarrow_\Sigma \psi}{\Gamma; \Delta, \mathbf{1} \longrightarrow_\Sigma \psi}\ \mathbf{1}\mathsf{L} \qquad \frac{}{\Gamma; \cdot \longrightarrow_\Sigma \mathbf{1}}\ \mathbf{1}\mathsf{R}$$

$$\frac{\Gamma; \Delta, \varphi_1, \varphi_2 \longrightarrow_\Sigma \psi}{\Gamma; \Delta, \varphi_1 \otimes \varphi_2 \longrightarrow_\Sigma \psi}\ \otimes\mathsf{L} \qquad \frac{\Gamma; \Delta_1 \longrightarrow_\Sigma \varphi_1 \quad \Gamma; \Delta_2 \longrightarrow_\Sigma \varphi_2}{\Gamma; \Delta_1, \Delta_2 \longrightarrow_\Sigma \varphi_1 \otimes \varphi_2}\ \otimes\mathsf{R}$$

$$\frac{\Gamma; \Delta_1 \longrightarrow_\Sigma \varphi_1 \quad \Gamma; \Delta_2, \varphi_2 \longrightarrow_\Sigma \psi}{\Gamma; \Delta_1, \Delta_2, \varphi_1 \multimap \varphi_2 \longrightarrow_\Sigma \psi}\ \multimap\mathsf{L} \qquad \frac{\Gamma; \Delta, \varphi_1 \longrightarrow_\Sigma \varphi_2}{\Gamma; \Delta \longrightarrow_\Sigma \varphi_1 \multimap \varphi_2}\ \multimap\mathsf{R}$$

$$\frac{\Gamma, \varphi; \Delta \longrightarrow_\Sigma \psi}{\Gamma; \Delta, !\varphi \longrightarrow_\Sigma \psi}\ !\mathsf{L} \qquad \frac{\Gamma; \cdot \longrightarrow_\Sigma \varphi}{\Delta; \cdot \longrightarrow_\Sigma !\varphi}\ !\mathsf{R}$$

$$\frac{\cdot \vdash_\Sigma t : \tau \quad \Gamma; \Delta, [t/x]\varphi \longrightarrow_\Sigma \psi}{\Gamma; \Delta, \forall x : \tau.\, \varphi \longrightarrow_\Sigma \psi}\ \forall\mathsf{L} \qquad \frac{\Gamma; \Delta \longrightarrow_{\Sigma, x:\tau} \varphi}{\Gamma; \Delta \longrightarrow_\Sigma \forall x : \tau.\, \varphi}\ \forall\mathsf{R}$$

$$\frac{\Gamma; \Delta, \varphi \longrightarrow_{\Sigma, x:\tau} \psi}{\Gamma; \Delta, \exists x : \tau.\, \varphi \longrightarrow_\Sigma \psi}\ \exists\mathsf{L} \qquad \frac{\cdot \vdash_\Sigma t : \tau \quad \Gamma; \Delta \longrightarrow_\Sigma [t/x]\varphi}{\Gamma; \Delta \longrightarrow_\Sigma \exists x : \tau.\, \varphi}\ \exists\mathsf{R}$$

Figure B.1: Multiplicative Exponential Intuitionistic Linear Logic — MEILL

B.2 Focused MEILL

A focusing presentation of logic hinges on the observation that some sequent rules are invertible, i.e., their application during proof search can always be "undone", while other rules are not. Invertible rules can therefore be applied eagerly and exhaustively until a *stable sequent* emerges — a sequent where no such rules are applicable. In well-behaved logics, in particular traditional and linear logic, non-invertible rules can be chained without compromising completeness. Chaining means that, when applying a non-invertible rule, the next non-invertible rules target just the by-products in the rule's premises of the decomposition of the principal formula in its conclusion. Completeness ensures that this can be carried out for as long as non-invertible rules can be applied to these by-products.

This suggests a proof-search strategy that alternates two phases: an *inversion phase* where invertible rules are applied exhaustively. A formula in the resulting stable sequent is then selected as the focus of the search. A *chaining phase* applies non-invertible rules to the formula in focus exclusively, putting its immediate subformulas in focus in its premises. Chaining proceeds until no non-invertible rule is applicable to the formula in focus. At this point, a new inversion phase begins, and so on until the derivation is completed or a failure is reported. In the latter case, proof search backtracks to the beginning of the last chaining phase, where it selects another formula and repeats the process. This approach to proof search is called *focusing*, and a logic for which it is complete is called focused (every logic for which chaining is complete is focused).

A focused presentation of linear logic relies on the observation that either the left rules or the right rules of each logical operator is invertible. This suggests classifying formulas accordingly. Positive formulas φ^+ have invertible left rules and negative formulas φ^- have invertible right rules. One of the most remarkable facts about focusing is that we can arbitrarily classify atomic formulas as positive or negative without impacting derivability, although the choice determines the shape of derivations. This classification effectively assigns polarities to each logical operator and to atoms. This leads to a polarized presentation of logic. A polarized grammar for MEILL is as follows:

$$\begin{array}{rlll} \textit{Positive formulas} & \varphi^+ & ::= & A^+ \mid \mathbf{1} \mid \varphi \otimes \varphi \mid\ !\varphi \mid \exists x : \tau.\, \varphi \\ \textit{Negative formulas} & \varphi^- & ::= & A^- \mid \varphi \multimap \varphi \mid \forall x : \tau.\, \varphi \\ \textit{Formulas} & \varphi & ::= & \varphi^+ \mid \varphi^- \end{array}$$

Forward-chaining search, which is what $\mathcal{L}^{1.5}$ relies on, is obtained by making all atoms positive. This is why we have grayed out negative atoms in this grammar.

A stable sequent is one where the linear context consists solely of negative formulas or positive atoms, and the right-hand side is either a positive formula or a negative atom (this latter case does not apply in our setting).

$$\begin{array}{rlll} \text{Stable linear context} & \Delta & ::= & \cdot \mid \Delta, \varphi^- \mid \Delta, A^+ \\ \text{Stable right-hand side} & \psi & ::= & \varphi^+ \mid A^- \end{array}$$

A focused presentation of MEILL (and linear logic in general) relies on the following four judgments:

$$\begin{array}{ll} \Gamma; \Delta \Longrightarrow_\Sigma \varphi & \text{Generic sequent} \\ \Gamma; \Delta \Longrightarrow_\Sigma \psi & \text{Stable sequent} \\ \Gamma; \Delta, \boxed{\varphi} \Longrightarrow_\Sigma \psi & \text{Left-focused sequent} \\ \Gamma; \Delta \Longrightarrow_\Sigma \boxed{\psi} & \text{Right-focused sequent} \end{array}$$

A generic sequent is subject to the application of invertible rules until a stable sequent is produced. Then, a formula on either the left or the right is selected as the focus and non-invertible rules are applied to it and its descendant for as long as possible. The focus is highlighted in red.

These judgments are realized by the rules in Figure B.2, where again we have grayed out rules that do not apply in our setting. Notice that again, aside from $\otimes$ and $\mathbf{1}$, only left rules are retained. Once more, we omit the cut rules, which are admissible.

The lines marked "focus" and "blur" are where phase alternation takes place. Specifically each of the "focus" rules selects a formula from a stable sequent to start the chaining phase. The "blur" rules terminate the chaining phase once no non-invertible rule is applicable to the formula in focus, thereby starting a new inversion phase. Observe that the above rules maintain the invariant that each focused sequent has exactly one formula in focus.

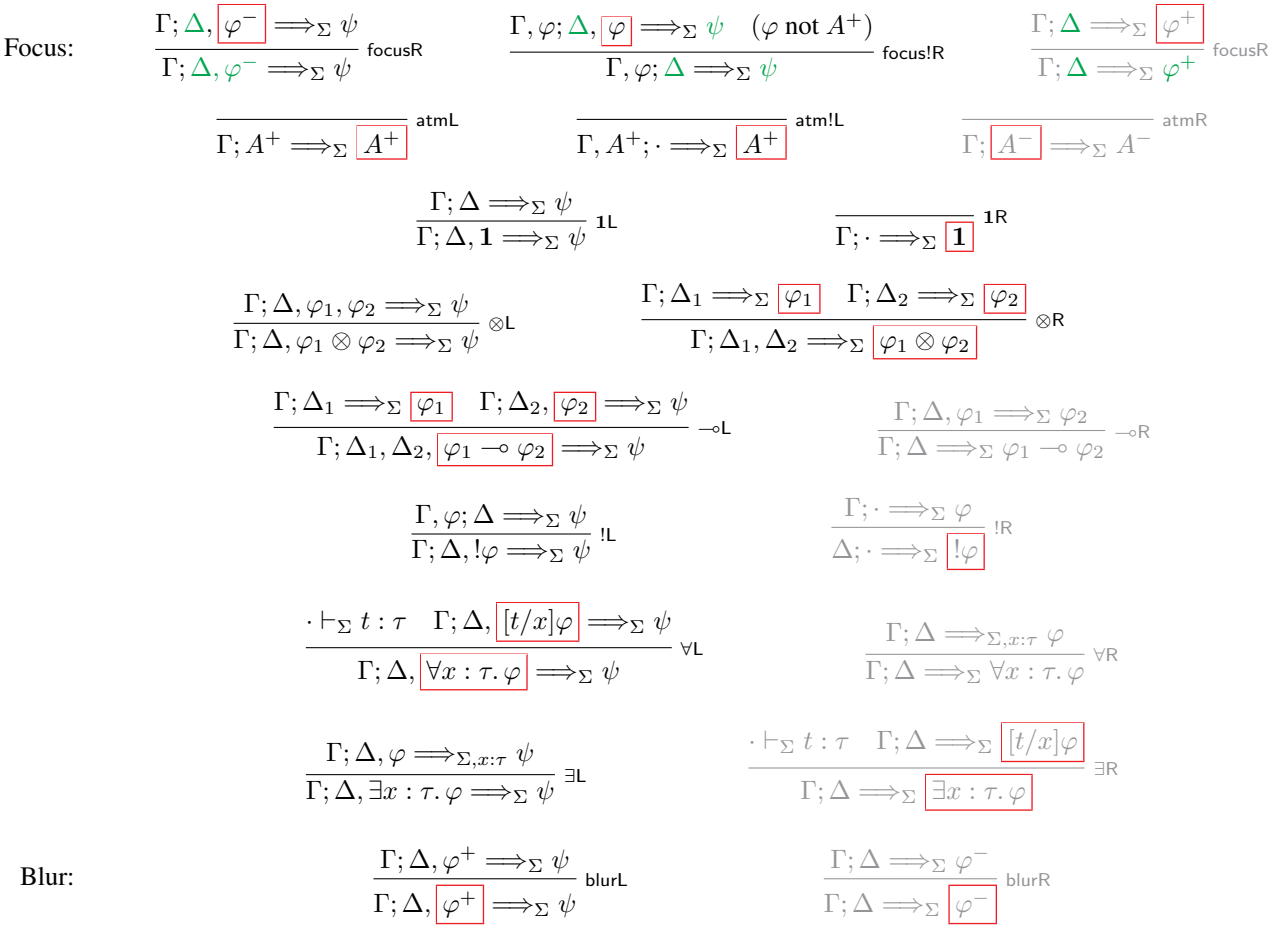


Figure B.2: Focused Presentation of MEILL

Focusing is sound and complete for intuitionistic linear logic as shown in [22]. Worth mentioning is also the elegant structural proof in [35] — it applies to traditional rather than linear logic although nothing prevents porting it to the linear setting.

Theorem 7 (Soundness and Completeness of Focusing) $\Gamma; \Delta \longrightarrow_\Sigma \psi$ *if and only if* $\Gamma; \Delta \Longrightarrow_\Sigma \psi$.

Proof See [22, 35]. □

Formula templates that, when focused upon, are processed through non-invertible rules only (i.e., that have derivation stubs consisting of a chain of non-invertible rules) can be viewed as *synthetic connectives*. This derivation stub coalesces into derived (non-invertible) rule, a right rule if the chain starts on the right and a left rule if it starts on the left. It turns out that this formula template can be decomposed by means of invertible rules only when put on the other side of the sequent. The corresponding derived rule is therefore itself invertible.

B.3 Interpretation

In this section, we identify $\mathcal{L}^{1.5}$ as a fragment of MEILL, show that its unfocused rewriting semantics stems from the traditional derivation rules of MEILL for that fragment, and that its focused transitions are explained by the corresponding focused rules.

B.3.1 Translation

The rewriting language $\mathcal{L}^{1.5}$ examined in this report corresponds to a fragment of MEILL. Atoms A of $\mathcal{L}^{1.5}$ are exactly the atomic formulas A of MEILL. The quantifiers, ! and the rewrite directive $\multimap$ are mapped to the identical operators of linear logic. Multiset union "," and the empty multiset "$\cdot$" within a program correspond instead to the tensor $\otimes$ and its unit $\mathbf{1}$.

This correspondence is formalized as follows, where the MEILL formula for program P is denoted $\ulcorner P \urcorner$, and similarly for left-hand sides and rules.

	Rewriting		**Logic**
LHS	$\ulcorner \cdot \urcorner$	$=$	$\mathbf{1}$
	$\ulcorner A, l \urcorner$	$=$	$A \otimes \ulcorner l \urcorner$
Rules	$\ulcorner l \multimap P \urcorner$	$=$	$\ulcorner l \urcorner \multimap \ulcorner P \urcorner$
	$\ulcorner \forall x{:}\,\iota.\, P \urcorner$	$=$	$\forall x{:}\,\iota.\, \ulcorner P \urcorner$
	$\ulcorner \forall X{:}\,\tau \to o.\, P \urcorner$	$=$	$\forall X{:}\,\tau \to o.\, \ulcorner P \urcorner$
Programs	$\ulcorner \cdot \urcorner$	$=$	$\mathbf{1}$
	$\ulcorner P_1, P_2 \urcorner$	$=$	$\ulcorner P_1 \urcorner \otimes \ulcorner P_2 \urcorner$
	$\ulcorner A \urcorner$	$=$	A
	$\ulcorner !A \urcorner$	$=$	$!\ulcorner A \urcorner$
	$\ulcorner R \urcorner$	$=$	$\ulcorner R \urcorner$
	$\ulcorner !R \urcorner$	$=$	$!\ulcorner R \urcorner$
	$\ulcorner \exists x{:}\,\tau \to \iota.\, P \urcorner$	$=$	$\exists x{:}\,\tau \to \iota.\, \ulcorner P \urcorner$
	$\ulcorner \exists X{:}\,\tau \to o.\, P \urcorner$	$=$	$\exists X{:}\,\tau \to o.\, \ulcorner P \urcorner$

Here, we have made explicit the distinction between first- and second-order entities in MEILL since we distinguished them in $\mathcal{L}^{1.5}$.

This translation identifies a linguistic fragment of MEILL that is given by the following grammar:

$$
\begin{array}{rlll}
\textit{LHS} & \varphi_l & ::= & \cdot \mid A \otimes \varphi_l \\
\textit{Rules} & \varphi_R & ::= & \varphi_l \multimap \varphi_P \mid \forall x : \iota.\, \varphi_R \\
 & & \mid & \forall X : \tau \to o.\, \varphi_R \\
\textit{Programs} & \varphi_P & ::= & \cdot \mid \varphi_P \otimes \varphi_P \mid A \mid\ !A \mid\ \varphi_R \mid\ !\varphi_R \mid\ \exists x : \tau \to \iota.\, \varphi_P \\
 & & \mid & \exists X : \tau \to o.\, \varphi
\end{array}
$$

This is nothing more than the grammar of left-hand sides, rules and programs from Appendix A.1 expressed using MEILL formulas.

We also give an encoding of archives to linear formulas consisting of the conjunction of their constituents, as we rely on these entities to discuss the correspondence between $\mathcal{L}^{1.5}$ and MEILL.

	Rewriting		**Logic**
Archives	$\ulcorner \cdot \urcorner$	$=$	$\mathbf{1}$
	$\ulcorner \Omega, A \urcorner$	$=$	$\ulcorner \Omega \urcorner \otimes\ !A$
	$\ulcorner \Omega, R \urcorner$	$=$	$\ulcorner \Omega \urcorner \otimes\ !\ulcorner R \urcorner$

Then, given a state $\Sigma.\langle \Omega \,;\, \Pi \rangle$, we write $\exists\Sigma.\ulcorner !\Omega; \Pi \urcorner$ for the MEILL formula $\exists\Sigma.\ulcorner \Omega \urcorner \otimes \ulcorner \Pi \urcorner$ obtained by prefixing the combined translation $\ulcorner \Omega \urcorner \otimes \ulcorner \Pi \urcorner$ of Ω and Π with an existential quantifier for every declaration in the signature Σ.

Below, we will make use of the following homomorphic mapping of an archive Ω to (persistent) contexts Γ in MEILL:

	Rewriting		**Logic**
Archives	$\lceil \cdot \rceil$	$=$	$\cdot$
	$\lceil \Omega, A \rceil$	$=$	$\lceil \Omega \rceil, A$
	$\lceil \Omega, R \rceil$	$=$	$\lceil \Omega \rceil, \lceil R \rceil$

We will also rely on the following non-deterministic encoding of the program component Π of a state $\Sigma.\langle \Omega \ ; \ \Pi \rangle$ to a (linear) context Δ:

	Rewriting		**Logic**
State programs	$\lceil P_1, P_2 \rceil$	$=$	$\lceil P_1 \rceil, \lceil P_2 \rceil$
	$\lceil P \rceil$	$=$	$\ulcorner P \urcorner$

This encoding is non-deterministic as the union (P_1, P_2) of a two $\mathcal{L}^{1.5}$ programs can be translated either a context that contains a single formula $\ulcorner P_1, P_2 \urcorner$ or as a context which contains possible more formulas obtained by encoding P_1 and P_2 in the same way recursively.

B.3.2 Unfocused Transitions

State transition rules in $\mathcal{L}^{1.5}$ correspond to derivable sequents of MEILL, a result that has been proved in a more general setting in [9]. In fact, each of our transition rules stems from one of the left rules of the operators in Appendix B.1, possibly helped by some other rules that play a structural role. The transitions for the rewriting quantifiers match exactly the left rules for the logic quantifiers when read from conclusion to premise. The transition for $\multimap$ stems from rule $\multimap$ L, relying on the rules for $\otimes$ and $\mathbf{1}$ to break the corresponding elements in the state program and rules init and clone to match each atom within with a corresponding left-hand side atom.

Given a state $\Sigma.\langle \Omega \ ; \ \Pi \rangle$, the following property is proved as in [9].

Theorem 8 (Soundness) *If* $\vdash \Sigma.\langle \Omega \ ; \ \Pi \rangle$ *state and* $\Sigma.\langle \Omega \ ; \ \Pi \rangle \mapsto \Sigma'.\langle \Omega' \ ; \ \Pi' \rangle$*, then* $\lceil \Omega \rceil; \lceil \Pi \rceil \longrightarrow_{\Sigma} \exists \Sigma'. \ulcorner !\Omega'; \Pi' \urcorner$.

While a formal proof can be found in [9], we now show the derivation snippet that corresponds to each transition. As we do so, we make implicit use of the left derivation rules for $\otimes$ and $\mathbf{1}$ to break $\lceil \Pi \rceil$ into context formulas that do not have these connectives as their topmost operator. The mapping between unfocused transitions in $\mathcal{L}^{1.5}$ and derivation snippets in linear logic is as follows, where the formula ψ in each case is as in the above theorem [9] and we are

making implicit uses of the definition of the encodings $\ulcorner - \urcorner$ and $\lceil - \rceil$.

$$\Sigma.\langle \Omega, l_1 \,;\, \Pi, l_2, (l_1, l_2) \multimap P \rangle \mapsto \Sigma.\langle \Omega, l_1 \,;\, \Pi, P \rangle \quad \rightsquigarrow \quad \frac{\cdots \quad \overline{\lceil \Omega, l_1 \rceil; A \longrightarrow_\Sigma A} \quad \cdots \quad \lceil \Omega, l_1 \rceil; \lceil l_2 \rceil \longrightarrow_\Sigma \ulcorner l_1, l_2 \urcorner \quad \lceil \Omega, l_1 \rceil; \lceil \Pi, P \rceil \longrightarrow_\Sigma \psi}{\lceil \Omega, l_1 \rceil; \lceil \Pi, l_2, (l_1, l_2) \multimap P \rceil \longrightarrow_\Sigma \psi} \multimap\mathsf{L}$$

$$\Sigma.\langle \Omega \,;\, \Pi, \forall x : \iota.\, R \rangle \mapsto \Sigma.\langle \Omega \,;\, \Pi, [t/x]R \rangle \quad \text{if } \cdot \vdash_\Sigma t : \iota \quad \rightsquigarrow \quad \frac{\cdot \vdash_\Sigma t : \iota \quad \lceil \Omega \rceil; \lceil \Pi, [t/x]R \rceil \longrightarrow_\Sigma \psi}{\lceil \Omega \rceil; \lceil \Pi, \forall x : \iota.\, R \rceil \longrightarrow_\Sigma \psi} \forall\mathsf{L}$$

$$\Sigma.\langle \Omega \,;\, \Pi, \forall X : \tau \to o.\, R \rangle \mapsto \Sigma.\langle \Omega \,;\, \Pi, [p/X]R \rangle \quad \text{if } p : \tau \to o \text{ in } \Sigma \quad \rightsquigarrow \quad \frac{p : \tau \to o \text{ in } \Sigma \quad \lceil \Omega \rceil; \lceil \Pi, [p/X]R \rceil \longrightarrow_\Sigma \psi}{\lceil \Omega \rceil; \lceil \Pi, \forall X : \tau \to o.\, R \rceil \longrightarrow_\Sigma \psi} \forall\mathsf{L}$$

$$\Sigma.\langle \Omega \,;\, \Pi, !A \rangle \mapsto \Sigma.\langle \Omega, A \,;\, \Pi \rangle \quad \rightsquigarrow \quad \frac{\lceil \Omega, A \rceil; \lceil \Pi \rceil \longrightarrow_\Sigma \psi}{\lceil \Omega \rceil; \lceil \Pi, !A \rceil \longrightarrow_\Sigma \psi} !\mathsf{L}$$

$$\Sigma.\langle \Omega \,;\, \Pi, !R \rangle \mapsto \Sigma.\langle \Omega, R \,;\, \Pi \rangle \quad \rightsquigarrow \quad \frac{\lceil \Omega, R \rceil; \lceil \Pi \rceil \longrightarrow_\Sigma \psi}{\lceil \Omega \rceil; \lceil \Pi, !R \rceil \longrightarrow_\Sigma \psi} !\mathsf{L}$$

$$\Sigma.\langle \Omega \,;\, \Pi, \exists x : \tau \to \iota.\, P \rangle \mapsto (\Sigma, x : \tau \to \iota).\langle \Omega \,;\, \Pi, P \rangle \quad \rightsquigarrow \quad \frac{\lceil \Omega \rceil; \lceil \Pi, P \rceil \longrightarrow_{\Sigma, x:\tau\to\iota} \psi}{\lceil \Omega \rceil; \lceil \Pi, \exists x : \tau \to \iota.\, P \rceil \longrightarrow_\Sigma \psi} \exists\mathsf{L}$$

$$\Sigma.\langle \Omega \,;\, \Pi, \exists X : \tau \to o.\, P \rangle \mapsto (\Sigma, X : \tau \to o).\langle \Omega \,;\, \Pi, P \rangle \quad \rightsquigarrow \quad \frac{\lceil \Omega \rceil; \lceil \Pi, P \rceil \longrightarrow_{\Sigma, X:\tau\to o} \psi}{\lceil \Omega \rceil; \lceil \Pi, \exists X : \tau \to o.\, P \rceil \longrightarrow_\Sigma \psi} \exists\mathsf{L}$$

$$\Sigma.\langle \Omega, R \,;\, \Pi \rangle \mapsto \Sigma.\langle \Omega, R \,;\, \Pi, R \rangle \quad \rightsquigarrow \quad \frac{\lceil \Omega, R \rceil; \lceil \Pi, R \rceil \longrightarrow_\Sigma \psi}{\lceil \Omega, R \rceil; \lceil \Pi \rceil \longrightarrow_\Sigma \psi} \mathsf{clone}$$

The first correspondence, for $\multimap$, requires some explanation: the derivation on the left branch is completed by using rules $\otimes\mathsf{L}$ and $\mathbf{1}\mathsf{L}$ to decompose $\lceil l_2 \rceil$ into individual atomic formulas in the linear context. Then, rules $\otimes\mathsf{R}$ and $\mathbf{1}\mathsf{R}$ are applied to break the formula $\ulcorner l_1, l_2 \urcorner$ into atoms. Doing so exhaustively leaves sequents of either the form $\lceil \Omega, l_1 \rceil; A \longrightarrow_\Sigma A$ or $\lceil \Omega, l_1', A \rceil; \cdot \longrightarrow_\Sigma A$ where $l_1 = l_1', A$. The first form is immediately resolved by a use of rule init, the second prepares for such an application by means of rule clone.

It is worth noting that this proof does not makes use of the grayed out rules from Appendix B.1. However, were we to replace the initial rule

$$\frac{}{\Gamma; \varphi \longrightarrow_\Sigma \varphi} \mathsf{init} \qquad \text{with the more canonical} \qquad \frac{}{\Gamma; A \longrightarrow_\Sigma A} \mathsf{atm}$$

the resulting derivation would need the grayed out rules, in general.

The above soundness theorem holds for the iterated transition relation $\mapsto^*$.

B.3.3 Focused Transitions

Accounting for polarities in the translation of $\mathcal{L}^{1.5}$ we just saw, left-hand sides and rules yield negative formulas while programs map to positive formulas:

$$\begin{array}{rrcl}
\textit{LHS} & \varphi_l^+ & ::= & \cdot \mid A^+ \otimes \varphi_l^+ \\
\textit{Rules} & \varphi_R^- & ::= & \varphi_l \multimap \varphi_P^+ \mid \forall x : \iota.\, \varphi_R^- \\
 & & \mid & \forall X : \tau \to o.\, \varphi_R^- \\
\textit{Programs} & \varphi_P^+ & ::= & \cdot \mid \varphi_P^+ \otimes \varphi_P^+ \mid A^+ \mid\ !A^+ \mid \varphi_R^- \mid\ !\varphi_R^- \mid \exists x : \tau \to \iota.\, \varphi_P^+ \\
 & & \mid & \exists X : \tau \to o.\, \varphi
\end{array}$$

The lone occurrence of φ_R^- in the production for φ_P^+ can be accounted for by rewriting it as $\varphi_R^- \otimes \mathbf{1}$, for example

Observe that a program state, which we wrote Π, consists of atoms A^+, negative conjunctions $\varphi_P^+ \otimes \varphi_P^+$ and units $\mathbf{1}$ (which can be simplified immediately as the left rules of these operators are invertible), existentials (whose left rules are also invertible), exponentials (whose left rule is invertible too), and formulas corresponding to either rules φ_R^- or atoms A^+. A stable state corresponds to a stable program: it contains only atoms and the representation of single-use rules only. This is also the form the persistent context has — it is populated from an archive.

The focused transition rules discussed in Appendix A.8 match focused derivation rules of MEILL in a more sophisticated way than their unfocused counterpart. The transitions for the existential programs and reusable entities work in the same way: existential and exponential formulas are positive since the left rule of their operator is invertible. Rules $\forall(l \multimap P)$ in $\mathcal{L}^{1.5}$ correspond however to negative formulas according to our translations. Therefore they start a chaining phase which ends when the (translation of) the embedded program P (a positive formula) is processed. The rewriting transition for $\mathcal{L}^{1.5}$ rules captures exactly this chaining phase. Reusable $\mathcal{L}^{1.5}$ rules are handled similarly, but their use is introduced by rule focus!L.

The program component Π of a stable state is transformed into a linear context obtained by mapping all occurrences of "," and "·" to their identically-denoted context-level operators. Therefore, a stable state is mapped to a context consisting solely of single-use atoms and rules. It is defined as follows.

	Rewriting		**Logic**
Stable state programs	$\lceil \cdot \rceil$	$=$	$\cdot$
	$\lceil \Pi, A \rceil$	$=$	$\lceil \Pi \rceil, A^+$
	$\lceil \Pi, R \rceil$	$=$	$\lceil \Pi \rceil, \ulcorner R \urcorner$

The above argument is summarized by the following soundness theorem, which distinguishes cases based on whether the starting state is stable or not (after applying rules $\otimes$L and $\mathbf{1}$L exhaustively).

Theorem 9 (Soundness) *Assume that* $\vdash \Sigma.\langle \Omega \,;\, \Pi \rangle$ *state.*

1. *If* Π *is stable (i.e.,* Π *has the form* Π*) and* $\Sigma.\langle \Omega \,;\, \Pi \rangle \Mapsto \Sigma'.\langle \Omega' \,;\, \Pi' \rangle$*, then* $\lceil \Omega \rceil; \lceil \Pi \rceil \Longrightarrow_\Sigma \exists \Sigma'. \ulcorner !\Omega'; \Pi' \urcorner$.
2. *If* Π *is not stable and* $\Sigma.\langle \Omega \,;\, \Pi \rangle \Mapsto \Sigma'.\langle \Omega' \,;\, \Pi' \rangle$*, then* $\lceil \Omega \rceil; \lceil \Pi \rceil \Longrightarrow_\Sigma \exists \Sigma'. \ulcorner !\Omega'; \Pi' \urcorner$.

Proof The proof proceeds as in the unfocused case, except for an additional induction on the number of universal quantifiers appearing in an $\mathcal{L}^{1.5}$ rule. □

It should be observed that the constructed derivation can make use of all the rules of focused linear logic, including the ones that have been grayed out. This is the reason why focused and unfocused transitions in $\mathcal{L}^{1.5}$ are not equivalent, despite Theorem 7.

As in the unfocused case, we distill the key elements of the proof by showing how each focused transition in $\mathcal{L}^{1.5}$

maps to a focused derivation snippet in MEILL.

$$\Sigma.\langle \Omega, l_1\theta \,;\, \Pi, l_2\theta, \forall(l_1, l_2 \multimap P)\rangle \;\Mapsto\; \Sigma.\langle \Omega, l_1\theta \,;\, \Pi, P\theta\rangle \quad \rightsquigarrow$$

$$\dfrac{\dfrac{\cdots \quad \dfrac{}{\lceil\Omega, l_1\theta\rceil; \lceil A\theta\rceil \Longrightarrow_\Sigma \boxed{A^+\theta}}\,\mathsf{atmL} \quad \cdots \quad \dfrac{}{\lceil\Omega, l_1'\theta, A\theta\rceil; \cdot \Longrightarrow_\Sigma \boxed{A^+\theta}}\,\mathsf{atm!L} \quad \cdots}{\lceil\Omega, l_1\theta\rceil; \lceil l_2\theta\rceil \Longrightarrow_\Sigma \boxed{\ulcorner l_1\theta, l_2\theta\urcorner}} \qquad \dfrac{\lceil\Omega, l_1\theta\rceil; \lceil\Pi, P\theta\rceil \Longrightarrow_\Sigma \psi}{\lceil\Omega, l_1\theta\rceil; \lceil\Pi\rceil, \boxed{\ulcorner P\theta\urcorner} \Longrightarrow_\Sigma \psi}\,\mathsf{blurL}}{\lceil\Omega, l_1\theta\rceil; \lceil\Pi, l_2\theta\rceil, \boxed{\ulcorner l_1, l_2 \multimap P\urcorner\theta} \Longrightarrow_\Sigma \psi}\,\multimap\mathsf{L}$$

$$\dfrac{\lceil\Omega, l_1\theta\rceil; \lceil\Pi, l_2\theta\rceil, \boxed{\ulcorner \forall(l_1, l_2 \multimap P)\urcorner} \Longrightarrow_\Sigma \psi}{\lceil\Omega, l_1\theta\rceil; \lceil\Pi, l_2\theta, \forall(l_1, l_2 \multimap P)\rceil \Longrightarrow_\Sigma \psi}\,\mathsf{focusL}$$

$$\Sigma.\langle \underbrace{\Omega, l_1\theta, \forall(l_1, l_2 \multimap P)}_{\Omega^*} \,;\, \Pi, l_2\theta\rangle \;\Mapsto\; \Sigma.\langle \Omega^* \,;\, \Pi, P\theta\rangle \quad \rightsquigarrow \quad \dfrac{\lceil\Omega^*\rceil; \lceil\Pi, l_2\theta\rceil, \boxed{\ulcorner \forall(l_1, l_2 \multimap P)\urcorner} \Longrightarrow_\Sigma \psi}{\lceil\Omega^*\rceil; \lceil\Pi, l_2\theta\rceil \Longrightarrow_\Sigma \psi}\,\mathsf{focus!L}$$

(as above)

$$\Sigma.\langle \Omega \,;\, \Pi, !A\rangle \;\Mapsto\; \Sigma.\langle \Omega, A \,;\, \Pi\rangle \quad \rightsquigarrow \quad \dfrac{\lceil\Omega, A\rceil; \lceil\Pi\rceil \Longrightarrow_\Sigma \psi}{\lceil\Omega\rceil; \lceil\Pi, !A\rceil \Longrightarrow_\Sigma \psi}\,\mathsf{clone}$$

$$\Sigma.\langle \Omega \,;\, \Pi, !R\rangle \;\Mapsto\; \Sigma.\langle \Omega, R \,;\, \Pi\rangle \quad \rightsquigarrow \quad \dfrac{\lceil\Omega, R\rceil; \lceil\Pi\rceil \Longrightarrow_\Sigma \psi}{\lceil\Omega\rceil; \lceil\Pi, !R\rceil \Longrightarrow_\Sigma \psi}\,\mathsf{clone}$$

$$\Sigma.\langle \Omega \,;\, \Pi, \exists x : \tau \to \iota.\, P\rangle \;\Mapsto\; (\Sigma, x : \tau \to \iota).\langle \Omega \,;\, \Pi, P\rangle \quad \rightsquigarrow \quad \dfrac{\lceil\Omega\rceil; \lceil\Pi, P\rceil \Longrightarrow_{\Sigma, x:\tau\to\iota} \psi}{\lceil\Omega\rceil; \lceil\Pi, \exists x : \tau \to \iota.\, P\rceil \Longrightarrow_\Sigma \psi}\,\exists\mathsf{L}$$

$$\Sigma.\langle \Omega \,;\, \Pi, \exists X : \tau \to o.\, P\rangle \;\Mapsto\; (\Sigma, X : \tau \to o).\langle \Omega \,;\, \Pi, P\rangle \quad \rightsquigarrow \quad \dfrac{\lceil\Omega\rceil; \lceil\Pi, P\rceil \Longrightarrow_{\Sigma, X:\tau\to o} \psi}{\lceil\Omega\rceil; \lceil\Pi, \exists X : \tau \to o.\, P\rceil \Longrightarrow_\Sigma \psi}\,\exists\mathsf{L}$$

Observe how chaining tracks exactly the structure of a $\mathcal{L}^{1.5}$ rule $\forall(l \multimap P)$. Because this pattern is fully handled by invertible rule when translated to logic, it gives rise to a synthetic connective, whose left rule has the form

$$\dfrac{\lceil\Omega, l_1\theta\rceil; \lceil\Pi\rceil, \boxed{\ulcorner P\theta\urcorner} \Longrightarrow_\Sigma \psi}{\lceil\Omega, l_1\theta\rceil; \lceil\Pi, l_2\theta\rceil, \boxed{\ulcorner \forall(l_1, l_2 \multimap P)\urcorner} \Longrightarrow_\Sigma \psi}$$

which corresponds closely to the focused transition for $\mathcal{L}^{1.5}$ rules. An exact correspondence is obtained by bracketing it between uses of rules focusL and blurL:

$$\dfrac{\lceil\Omega, l_1\theta\rceil; \lceil\Pi, P\theta\rceil \Longrightarrow_\Sigma \psi}{\lceil\Omega, l_1\theta\rceil; \lceil\Pi, l_2\theta, \forall(l_1, l_2 \multimap P)\rceil \Longrightarrow_\Sigma \psi}$$

The same happens to a reusable rule $!\forall(l \multimap P)$ once it has been archived (i.e., inserted in the persistent context using rule clone on the logic side). Bracketing the above synthetic rule between focus!L and blurL yields the derived rule

$$\dfrac{\lceil\Omega, l_1\theta, \forall(l_1, l_2 \multimap P)\rceil; \lceil\Pi, P\theta\rceil \Longrightarrow_\Sigma \psi}{\lceil\Omega, l_1\theta, \forall(l_1, l_2 \multimap P)\rceil; \lceil\Pi, l_2\theta\rceil \Longrightarrow_\Sigma \psi}$$

which corresponds exactly to the focused $\mathcal{L}^{1.5}$ transition for reusable rules.

Some of the same comments we made when discussing the unfocused correspondence apply here also. Additionally, the substitution term used for each instance of rule $\forall$L is exactly what the substitution θ associates to the variable processed by this rule. The proof of the sequent $\lceil\Omega, l_1\theta\rceil; \lceil l_2\theta\rceil \Longrightarrow_\Sigma \boxed{\ulcorner l_1\theta, l_2\theta\urcorner}$ is again interesting. Because the linear context $\ulcorner l_1\theta\urcorner$ is stable, it does not have occurrences of $\otimes$ nor $\mathbf{1}$ (it is a multiset of bare atoms). As a consequence, this derivation is chained all the way to the applications of rules atmL and atm!L (or $\mathbf{1}$R if l_2 is empty). As earlier, we wrote l_1 as l_1', A in the displayed use of rule atm!L.

The same argument can be made starting from the other formulations of the focused transition semantics of $\mathcal{L}^{1.5}$ seen in Appendix A.8. In particular, our second presentation, which decomposed the transitions for single-use and reusable rules into discrete steps, is essentially isomorphic to the focused left sequent rules for the logical operators corresponding to each case. Our third presentation, which went from stable state to stable state, combines the chaining phase of a step with the immediately following inversion phase.