Substructural Parametricity

C. B. Aberlé
Carnegie Mellon University, Pittsburgh, PA, USA

Karl Crary
Carnegie Mellon University, Pittsburgh, PA, USA

Chris Martens
Northeastern University, Boston, MA, USA

Frank Pfenning
Carnegie Mellon University, Pittsburgh, PA, USA

Abstract

Ordered, linear, and other substructural type systems allow us to expose deep properties of programs at the syntactic level of types. In this paper, we develop a family of unary logical relations that allow us to prove consequences of parametricity for a range of substructural type systems. A key idea is to parameterize the relation by an algebra, which we exemplify with a monoid and commutative monoid to interpret ordered and linear type systems, respectively. We prove the fundamental theorem of logical relations and apply it to deduce extensional properties of inhabitants of certain types. Examples include demonstrating that the ordered types for list append and reversal are inhabited by exactly one function, as are types of some tree traversals. Similarly, the linear type of the identity function on lists is inhabited only by permutations of the input. Our most advanced example shows that the ordered type of the list fold function is inhabited only by the fold function.

2012 ACM Subject Classification Theory of computation → Type structures

Keywords and phrases Substructural type systems, logical relations, ordered logic

Funding This material is based upon work supported by the United States Air Force Office of Scientific Research under grant number A210038S002 (Tristan Nguyen, program manager). Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the AFOSR.

Acknowledgements The authors thank Bob Atkey and Stephanie Balzer for helpful comments regarding related work.

1 Introduction

Substructural type systems and parametric polymorphism are two mechanisms for capturing precise behavioral properties of programs at the type level, enabling powerful static reasoning. The goal of this paper is to give a theoretical account of these mechanisms in combination.

Substructural type systems have been investigated since the advent of linear logic, starting with the seminal paper by Girard and Lafont [18]. Among other applications, with substructural type systems one can avoid garbage collection, update memory in place [27, 28], make message-passing [16, 12] or shared memory concurrency [17, 35] safe, model quantum computation [15], or reason efficiently about imperative programs [26]. Substructural type systems have thus been incorporated into languages that seek to offer such guarantees, such as Rust, Koka, Haskell, Oxidized OCaml, and ProtoQuipper.

Parametricity, originally introduced for System F [42], enables the idea that programs whose types involve universal quantification over type parameters have certain strong semantic properties. This idea supports powerful program reasoning principles such as representation independence across abstraction boundaries [30] and "theorems for free" that can be derived about all inhabitants of certain types, for example that every inhabitant of $\forall\alpha.\ \alpha \to \alpha$ is equivalent to the identity function [46].

The theory of substructural logics and type systems is now relatively well understood, including several ways to integrate substructural and structural type systems [9, 38, 19]. It is therefore somewhat surprising that we do not yet know much about how parametricity and its applications interact with them. The main foray into substructural parametricity is a paper by Zhao et al. [47] that accounts for a polymorphic dual-intuitionistic linear logic. They point out that logical relations on closed terms are problematic because substitution obscures linearity. Their solution was to construct a logical relation on open terms, necessitating the introduction of "semantic typing" judgments that mirror the syntactic type system, which complicates their definition and application.

In this paper, we follow an approach using *constructive resource semantics* in the style of Reed et al. [39, 41, 40] to construct logical relations on *closed terms*. We start with an ordered type system [37, 36, 24], which may be considered the least permissive among substructural type systems and therefore admits a pleasantly minimal definition. However, the construction is generic with respect to certain properties of the resource algebra, which allows us to extend it also to linear and unrestricted types. Consequences of our development include that certain polymorphic types are only inhabited by the polymorphic append and reverse functions on lists. Similarly, certain types are only inhabited by functions that swap or maintain the order of pairs. The most advanced application shows that the ordered type of fold over lists is inhabited only by the fold function.

We conjecture that the three substructural modes we investigate – ordered, linear, and unrestricted – can also be combined in an adjoint framework [9, 19] but leave this to future work. Similarly, we simplify our presentation by defining only a *unary* logical relation since it is sufficient to demonstrate proof-of-concept, but nothing stands in the way of a more general definition (for example, to support representation-independence results).

2 A Minimalist Fragment

We start with a small fragment of the Lambek Calculus [25, 29], extended with parametric polymorphism [43]. This fragment is sufficient to illustrate the main ideas behind our constructions. For the sake of simplicity we choose a Curry-style formulation of typing, concentrating on properties of untyped terms rather than intrinsically typed terms. This allows the same terms to inhabit ordered, linear, and unrestricted types and thereby focus on semantic rather than syntactic issues.

$$
\begin{array}{llcll}
\text{Types} & A, B & ::= & \alpha \mid \mathbf{1} \mid A \bullet B \mid A \rightarrowtail B \mid A \twoheadrightarrow B \mid \forall\alpha.\, A & \\
\text{Terms} & e & ::= & x & \\
 & & \mid & () \mid \mathbf{match}\ e\ (() \Rightarrow e') & (\mathbf{1}) \\
 & & \mid & (e_1, e_2) \mid \mathbf{match}\ e\ ((x, y) \Rightarrow e') & (A \bullet B) \\
 & & \mid & \lambda x.\ e \mid e_1\, e_2 & (A \rightarrowtail B, A \twoheadrightarrow B)
\end{array}
$$

In this fragment, we have $A \bullet B$ (read "A fuse B") which, logically, is a noncommutative conjunction. We have two forms of implication: $A \rightarrowtail B$ (read: "A under B", originally written as $A\backslash B$) which is true if from the hypothesis A *placed at the left end of the antecedents* we can deduce B, and $A \twoheadrightarrow B$ (read: "B over A", originally written as $B\ /\ A$) which is true if from the hypothesis A *placed at the right end of the antecedents* we can prove B. Lambek's original notation was suitable for the sequent calculus and its applications in linguistics, but is less readable for natural deduction and functional programming.

Our basic typing judgment has the form $\Delta \mid \Omega \vdash e : A$ where Δ consists of hypotheses α type, and Ω is an *ordered context* $(x_1 : A_1) \ldots (x_n : A_n)$. We make the standard presuppositions that $\Delta \vdash A$ type and $\Delta \vdash A_i$ type for every $x_i : A_i$ in Ω, and that both type variables and term variables are pairwise distinct. The rules are show in Figure 1.

$$\frac{}{\Delta \mid x : A \vdash x : A}\ \mathsf{hyp}$$

$$\frac{}{\Delta \mid \cdot \vdash () : \mathbf{1}}\ \mathbf{1}I \qquad \frac{\Delta \mid \Omega \vdash e : \mathbf{1} \quad \Delta \mid \Omega_L \Omega_R \vdash e' : C}{\Delta \mid \Omega_L \Omega \Omega_R \vdash \mathbf{match}\ e\ (() \Rightarrow e') : A}\ \mathbf{1}E$$

$$\frac{\Delta \mid \Omega\,(x : A) \vdash e : B}{\Delta \mid \Omega \vdash \lambda x.\, e : A \twoheadrightarrow B}\ {\twoheadrightarrow}I \qquad \frac{\Delta \mid \Omega \vdash e_1 : A \twoheadrightarrow B \quad \Delta \mid \Omega_A \vdash e_2 : A}{\Delta \mid \Omega\, \Omega_A \vdash e_1\, e_2 : B}\ {\twoheadrightarrow}E$$

$$\frac{\Delta \mid (x : A)\, \Omega \vdash e : B}{\Delta \mid \Omega \vdash \lambda x.\, e : A \rightarrowtail B}\ {\rightarrowtail}I \qquad \frac{\Delta \mid \Omega \vdash e_1 : A \rightarrowtail B \quad \Delta \mid \Omega_A \vdash e_2 : A}{\Delta \mid \Omega_A\, \Omega \vdash e_1\, e_2 : B}\ {\rightarrowtail}E$$

$$\frac{\Delta \mid \Omega_A \vdash e_1 : A \quad \Delta \mid \Omega_B \vdash e_2 : B}{\Delta \mid \Omega_A\, \Omega_B \vdash (e_1, e_2) : A \bullet B}\ {\bullet}I \qquad \frac{\Delta \mid \Omega \vdash e : A \bullet B \quad \Delta \mid \Omega_L\, (x : A)\, (y : B)\, \Omega_R \vdash e' : C}{\Delta \mid \Omega_L\, \Omega\, \Omega_R \vdash \mathbf{match}\ e\ ((x, y) \Rightarrow e') : C}\ {\bullet}E$$

$$\frac{\Delta, \alpha\ \mathsf{type} \mid \Omega \vdash e : A}{\Delta \mid \Omega \vdash e : \forall \alpha.\, A}\ \forall I \qquad \frac{\Delta \mid \Omega \vdash e : \forall \alpha. A(\alpha) \quad \Delta \vdash B\ \mathsf{type}}{\Delta \mid \Omega \vdash e : A(B)}\ \forall E$$

Figure 1 Ordered Natural Deduction.

Here are a few example judgments that hold or fail. We elide the context $\Delta = (\alpha\ \mathsf{type}, \beta\ \mathsf{type}, \gamma\ \mathsf{type})$.

	$\vdash$	$\lambda x.\, x : \alpha \rightarrowtail \alpha$	
	$\vdash$	$\lambda x.\, x : \alpha \twoheadrightarrow \alpha$	
	$\nvdash$	$\lambda x.\, \lambda y.\, x : \alpha \twoheadrightarrow (\beta \twoheadrightarrow \alpha)$	(no weakening)
	$\nvdash$	$\lambda x.\, (x, x) : \alpha \twoheadrightarrow (\alpha \bullet \alpha)$	(no contraction)
	$\vdash$	$\lambda x.\, \lambda y.\, (x, y) : \alpha \twoheadrightarrow (\beta \twoheadrightarrow (\alpha \bullet \beta))$	
	$\nvdash$	$\lambda x.\, \lambda y.\, (x, y) : \alpha \rightarrowtail (\beta \rightarrowtail (\alpha \bullet \beta))$	(no exchange)
$f : \beta \twoheadrightarrow (\alpha \rightarrowtail \gamma)$	$\vdash$	$\lambda x.\, \lambda y.\, (f\, y)\, x : \alpha \rightarrowtail (\beta \twoheadrightarrow \gamma)$	("associativity")
$g : \alpha \rightarrowtail (\beta \twoheadrightarrow \gamma)$	$\vdash$	$\lambda y.\, \lambda x.\, (g\, x)\, y : \beta \twoheadrightarrow (\alpha \rightarrowtail \gamma)$	
$g : (\alpha \bullet \beta) \twoheadrightarrow \gamma$	$\vdash$	$\lambda x.\, \lambda y.\, g\, (x, y) : \alpha \twoheadrightarrow (\beta \twoheadrightarrow \gamma)$	(currying)
$f : \alpha \twoheadrightarrow (\beta \twoheadrightarrow \gamma)$	$\vdash$	$\lambda p.\, \mathbf{match}\ p\ ((x, y) \Rightarrow f\, x\, y) : (\alpha \bullet \beta) \twoheadrightarrow \gamma$	(uncurrying)

The strictures of the typing judgment imply that certain types may be uninhabited, or may be inhabited by terms that are extensionally equivalent (i.e., that yield the same outputs for all closed inputs) to a small number of possibilities. To count the number of linear functions, translate $(A \twoheadrightarrow B)^{\mathsf{L}} = (A \rightarrowtail B)^{\mathsf{L}} = A^{\mathsf{L}} \multimap B^{\mathsf{L}}$ and $(A \bullet B)^{\mathsf{L}} = A^{\mathsf{L}} \otimes B^{\mathsf{L}}$ and similarly for unrestricted functions.

Types	Ordered	Linear	Unrestricted
$\alpha \twoheadrightarrow \alpha$	1	1	1
$\alpha \twoheadrightarrow (\alpha \twoheadrightarrow \alpha)$	0	0	2
$\alpha \twoheadrightarrow (\alpha \twoheadrightarrow (\alpha \bullet \alpha))$	1	2	4
$\alpha \twoheadrightarrow (\alpha \rightarrowtail (\alpha \bullet \alpha))$	1	2	4
$\alpha \twoheadrightarrow (\beta \twoheadrightarrow (\beta \bullet \alpha))$	0	1	1
$\alpha \twoheadrightarrow (\beta \twoheadrightarrow (\alpha \bullet \beta))$	1	1	1

Because our intended application language based on adjoint natural deduction [19] is call-by-value, we can give a straightforward big-step operational semantics [22] relating a term to its final value. Because this evaluation does not directly interact with or benefit from substructural properties, we show it without further comment in Figure 2. It has the property of preservation that if $\cdot \vdash e : A$ and $e \hookrightarrow v$ then $\cdot \vdash v : A$. Jang et al. give an account [19] that exploits linearity and other substructural properties, although not the lack of exchange.

$$\frac{}{() \hookrightarrow ()} \qquad \frac{e \hookrightarrow () \quad e' \hookrightarrow v}{\textbf{match}\ e\ (() \Rightarrow e') \hookrightarrow v}$$

$$\frac{}{\lambda x.\, e \hookrightarrow \lambda x.\, e} \qquad \frac{e_1 \hookrightarrow \lambda x.\, e_1' \quad e_2 \hookrightarrow v_2 \quad [v_2/x]e_1' \hookrightarrow v}{e_1\, e_2 \hookrightarrow v}$$

$$\frac{e_1 \hookrightarrow v_1 \quad e_2 \hookrightarrow v_2}{(e_1, e_2) \hookrightarrow (v_1, v_2)} \qquad \frac{e \hookrightarrow (v_1, v_2) \quad [v_1/x, v_2/y]e' \hookrightarrow v'}{\textbf{match}\ e\ ((x, y) \Rightarrow e') \hookrightarrow v'}$$

Figure 2 Big-Step Operational Semantics.

3 An Algebraic Logical Predicate

Because of our particular setting, we define two mutually dependent logical predicates: $[\![A]\!]$ for closed terms and $[A]$ for closed values. In addition, the relation is parameterized by elements from an algebraic domain which may have various properties. For the ordered case, it should be a monoid, for the linear case a commutative monoid. However, the rules themselves do not require this for the pure sets of terms. We use $m \cdot n$ for the binary operation on the monoid, and ϵ for its unit.

Ignoring polymorphism for now, we write $m \Vdash e \in [\![A]\!]$ and $m \Vdash v \in [A]$, which is defined by

$$\begin{array}{lcl}
m \Vdash e \in [\![A]\!] & \Longleftrightarrow & \exists v.\, e \hookrightarrow v \wedge m \Vdash v \in [A] \\[1ex]
m \Vdash v \in [\mathbf{1}] & \Longleftrightarrow & m = \epsilon \wedge v = (\,) \\
m \Vdash v \in [A \bullet B] & \Longleftrightarrow & \exists m_1, m_2.\, m = m_1 \cdot m_2 \wedge v = (v_1, v_2) \\
 & & \quad \wedge\, m_1 \Vdash v_1 \in [A] \wedge m_2 \Vdash v_2 \in [B] \\
m \Vdash v \in [A \twoheadrightarrow B] & \Longleftrightarrow & \forall k.\, k \Vdash w \in [A] \Longrightarrow m \cdot k \Vdash v\, w \in [\![B]\!] \\
m \Vdash v \in [A \rightarrowtail B] & \Longleftrightarrow & \forall k.\, k \Vdash w \in [A] \Longrightarrow k \cdot m \Vdash v\, w \in [\![B]\!]
\end{array}$$

We can see how the algebraic structure of the monoid tracks information about order if its operation is not commutative.

The key step, as usual in logical predicates of this nature, is the case for universal quantification and type variables. We map type variables α to relations R_B between monoid elements and values in $[B]$ where B is a closed type. We use S to denote this mapping from type variables to sets of values and write it as a superscript on $\Vdash$.

$$\begin{array}{lcl} m \Vdash^S v \in [\alpha] & \iff & m\ S(\alpha)\ v \\ m \Vdash^S v \in [\forall\alpha.\, A(\alpha)] & \iff & \forall B, R_B.\, m \Vdash^{S,\alpha\mapsto R_B} v \in [A(\alpha)] \end{array}$$

The mapping S is just passed through identically in the cases of the relation defined above.

We can already verify some interesting properties. As a first example we show that the logical predicates are nonempty.

▶ **Example 1.**

$$\epsilon \Vdash \lambda x.\, \lambda y.\, (x, y) \in [\![\forall\alpha.\, \alpha \twoheadrightarrow (\alpha \twoheadrightarrow (\alpha \bullet \alpha))]\!]$$

Proof. Because the λ-term is a value, we need to check

$$\epsilon \Vdash \lambda x.\, \lambda y.\, (x, y) \in [\forall\alpha.\, \alpha \twoheadrightarrow (\alpha \twoheadrightarrow (\alpha \bullet \alpha))]$$

By definition, this is true if for an arbitrary A and relation $m\ R_A\ v$ we have

$$\epsilon \Vdash^{\alpha\mapsto R_A} \lambda x.\, \lambda y.\, (x, y) \in [\alpha \twoheadrightarrow (\alpha \twoheadrightarrow (\alpha \bullet \alpha))]$$

Using the definition of the logical predicate for right implication twice and one intermediate step of evaluation, this holds iff

$$m \cdot k \Vdash^{\alpha\mapsto R_A} (\lambda y.\, (v, y))\ w \in [\![\alpha \bullet \alpha]\!]$$

for all m, k with $m \Vdash^{\alpha\mapsto R_A} v$ and $k \Vdash^{\alpha\mapsto R_A} w$. By evaluation, this is true iff

$$m \cdot k \Vdash^{\alpha\mapsto R_A} (v, w) \in [\alpha \bullet \alpha]$$

Now we can apply the definition of $[A \bullet B]$, splitting $m \cdot k$ into m and k and reducing it to

$$m \Vdash^{\alpha\mapsto R_A} v \in [\alpha] \wedge k \Vdash^{\alpha\mapsto R_A} w \in [\alpha]$$

Both of these hold because, by assumption, $m\ R_A\ v$ and $k\ R_A\ w$. ◀

More interesting, perhaps, is the reverse.

▶ **Example 2.** If

$$\epsilon \Vdash e \in [\![\forall\alpha.\, \alpha \twoheadrightarrow (\alpha \twoheadrightarrow (\alpha \bullet \alpha))]\!]$$

then e is extensionally equal to $\lambda x.\, \lambda y.\, (x, y)$, in that for any closed type A and closed terms $v, w : A$, it must be the case that $e\, v\, w \hookrightarrow (v, w)$. In particular, it can not be $\lambda x.\, \lambda y.\, (y, x)$.

Proof. We choose our monoid to be the free monoid over two generators a and b and we choose an arbitrary closed type A and two elements v and w. Moreoever, we pick R_A relating only $a\ R_A\ v$ and $b\ R_A\ w$.

From the definitions (and skipping over some simple properties regarding evaluation), we obtain

$$a \cdot b \Vdash^{\alpha\mapsto R_A} e\ v\ w \in [\![\alpha \bullet \alpha]\!]$$

By the clauses for $[\![\alpha \bullet \alpha]\!]$, $[\alpha \bullet \alpha]$ and α we conclude that

$$e\ v\ w \hookrightarrow (u_1, u_2)$$

for some values u_1 and u_2 with $a\ R_A\ u_1$ and $b\ R_A\ u_2$. Because the only value related to a is v and the only value related to b is w, we conclude $u_1 = v$ and $u_2 = w$. Therefore

$$e\ v\ w \hookrightarrow (v, w)$$

Since v and w were chosen arbitrarily, we see that e is extensionally equal to $\lambda x.\, \lambda y.\, (x, y)$. ◀

4 The Fundamental Theorem

The fundamental theorem of logical predicates states that every well-typed term is in the predicate. Our relations also include terms that are not well-typed, which can occasionally be useful when one exceeds the limits of static typing.

We need a few standard lemmas, adapted to this case. We only spell out one.

▶ **Lemma 3** (Compositionality). *Define R_A such that $k\ R_A\ w$ iff $k \Vdash w \in [A]$. Then $m \Vdash^{S,\alpha \mapsto R_A} v \in [B(\alpha)]$ iff $m \Vdash^{S} v \in [B(A)]$*

Proof. By induction on $B(\alpha)$. ◀

We would like to prove the fundamental theorem by induction over the structure of the typing derivation. Since our logical relation is defined for closed terms, we need a closing substitution η, i.e., a substitution of closed terms into the typing context of the term e for which we seek to prove the fundamental theorem by induction, such that the application $\eta(e)$ of η to e is a closed term. We therefore extend the definition of the logical relation to substitutions and contexts as follows:

$$\begin{array}{lcl} m \Vdash^S (x \mapsto v) \in [x : A] & \iff & m \Vdash^S v \in [A] \\ m \Vdash^S (\eta_1\, \eta_2) \in [\Omega_1\, \Omega_2] & \iff & \exists m_1, m_2.\, m = m_1 \cdot m_2 \wedge m_1 \Vdash^S \eta_1 \in [\Omega_1] \wedge m_2 \Vdash^S \eta_2 \in [\Omega_2] \\ m \Vdash^S (\cdot) \in [\cdot] & \iff & m = \epsilon \end{array}$$

Due to the associativity of the monoid operation and concatenation of contexts, this constitutes a valid definition.

▶ **Theorem 4** (Fundamental Theorem (purely ordered)). *Assume $\Delta \mid \Omega \vdash e : A$, a mapping S with domain Δ, and closing substitution $m \Vdash^S \eta \in [\Omega]$. Then $m \Vdash^S \eta(e) \in [\![A]\!]$.*

Proof. By induction on the structure of the given typing derivation. We show a few cases.
Case:

$$\frac{}{\Delta \mid x : A \vdash x : A}\ \mathsf{hyp}$$

Then $m \Vdash^S \eta(x) \in [A]$ by assumption and definition, and $m \Vdash^S \eta(x) \in [\![A]\!]$ since $\eta(x)$ is a value.

Case:

$$\frac{\Delta \mid \Omega\, (x : A) \vdash e : B}{\Delta \mid \Omega \vdash \lambda x.\, e : A \twoheadrightarrow B}\ {\twoheadrightarrow}I$$

$m \Vdash^S \eta \in [\Omega]$	Given
$k \Vdash^S v \in [A]$	Assumption (1)
$k \Vdash^S (x \mapsto v) \in [x : A]$	By definition
$m \cdot k \Vdash^S (\eta, x \mapsto v) \in [\Omega\,(x : A)]$	By definition
$m \cdot k \Vdash^S (\eta, x \mapsto v)(e) \in [\![B]\!]$	By ind. hyp.
$m \cdot k \Vdash^S (\eta(\lambda x.\, e))\, v \in [\![B]\!]$	By reverse evaluation, v closed
$m \Vdash^S \eta(\lambda x.\, e) \in [A \twoheadrightarrow B]$	By definition, discharging (1)
$m \Vdash^S \eta(\lambda x.\, e) \in [\![A \twoheadrightarrow B]\!]$	By definition

Case:

$$\frac{\Delta \mid \Omega \vdash e_1 : A \twoheadrightarrow B \quad \Delta \mid \Omega_A \vdash e_2 : A}{\Delta \mid \Omega\,\Omega_A \vdash e_1\, e_2 : B} \twoheadrightarrow E$$

$m \Vdash^S \eta \in [\Omega\,\Omega_A]$	Given
$m_1 \Vdash^S \eta_1 \in [\Omega]$ and $m_2 \Vdash^S \eta_2 \in [\Omega_A]$	
for some m_1, m_2, η_1, and η_2 with $m = m_1 \cdot m_2$ and $\eta = \eta_1\, \eta_2$	By definition
$m_1 \Vdash^S \eta_1(e_1) \in [\![A \twoheadrightarrow B]\!]$	By ind. hyp.
$m_2 \Vdash^S \eta_2(e_2) \in [\![A]\!]$	By ind. hyp.
$\eta_1(e_1) \hookrightarrow v_1$ with $m_1 \Vdash^S v_1 \in [A \twoheadrightarrow B]$	By definition
$\eta_2(e_2) \hookrightarrow v_2$ with $m_2 \Vdash^S v_2 \in [A]$	By definition
$m_1 \cdot m_2 \Vdash^S v_1\, v_2 \in [\![B]\!]$	By definition
$(\eta_1\, \eta_2)(e_1\, e_2) = (\eta_1(e_1))\,(\eta_2(e_2))$	By properties of substitution
$m \Vdash^S \eta(e_1\, e_2) \in [\![B]\!]$	Since $m = m_1 \cdot m_2$ and $\eta = (\eta_1\, \eta_2)$

Case:

$$\frac{\Delta, \alpha\ \mathsf{type} \mid \Omega \vdash e : A}{\Delta \mid \Omega \vdash e : \forall \alpha.\, A} \forall I$$

$m \Vdash^S \eta \in [\Omega]$	Given
R_B an arbitrary relation $k\ R_B\ v$	Assumption (1)
$m \Vdash^{S, \alpha \mapsto R_B} \eta \in [\Omega]$	Since α fresh
$m \Vdash^{S, \alpha \mapsto R_B} \eta(e) \in [\![A]\!]$	By ind. hyp.
$m \Vdash^S \eta(e) \in [\![\forall \alpha.\, A]\!]$	By definition, discharging (1)

Case:

$$\frac{\Delta \mid \Omega \vdash e : \forall \alpha. A(\alpha) \quad \Delta \vdash B\ \mathsf{type}}{\Delta \mid \Omega \vdash e : A(B)} \forall E$$

$m \Vdash^S \eta \in [\Omega]$	Given
$m \Vdash^S \eta(e) \in [\![\forall \alpha.\, A(\alpha)]\!]$	By ind. hyp.
Define $k\ R_B\ v$ iff $k \Vdash^S v \in [B]$	
$m \Vdash^{S, \alpha \mapsto R_B} \eta(e) \in [\![A(\alpha)]\!]$	By definition
$m \Vdash^{S, \alpha \mapsto R_B} v \in [A(\alpha)]$ for $\eta(e) \hookrightarrow v$	By definition
$m \Vdash^S v \in [A(B)]$	By compositionality (Lemma 3)
$m \Vdash^S \eta(e) \in [\![A(B)]\!]$	By definition

◀

Because typing implies that the logical predicate holds, the earlier examples now apply to well-typed terms.

▶ **Theorem 5** (Example 2 revisited)**.** *If*

$$\cdot \vdash e : \forall \alpha.\, \alpha \twoheadrightarrow (\alpha \twoheadrightarrow (\alpha \bullet \alpha))$$

then e is extensionally equivalent to $\lambda x.\, \lambda y.\, (x, y)$.

Proof. We just note that

$$\epsilon \Vdash e \in [\![\forall \alpha.\, \alpha \twoheadrightarrow (\alpha \twoheadrightarrow (\alpha \bullet \alpha))]\!]$$

since $(\cdot) \in [\cdot]$ and $(\cdot)e = e$ and the empty mapping S suffices without any free type variables. Then we appeal to the reasoning in Example 2. ◀

5 Unrestricted Functions

We are interested in properties of functions such as list append or list reversal, or higher-order functions such as fold. This requires inductive types, but the functions on them are not used linearly. For example, append has a recursive call in the case of a nonempty list, but none in the case of an empty list. We could introduce a general modality $!A$ for this purpose. A simpler alternative that is sufficient for our situation is to introduce unrestricted function types $A \to B$ (usually coded as $!A \multimap B$ in linear logic or $!A \twoheadrightarrow B$ in ordered logic). This path has been explored previously [37] with different motivations. There, an *open* logical relation was defined on the negative monomorphic fragment in order to show the existence of canonical forms, a property that is largely independent of ordered typing.

Adding unrestricted functions is rather straightforward in typing by using two kinds of variables: those that are ordered and those unrestricted. Then, in the logical predicate, unrestricted variables must not use any resources, that is, they are assigned the unit element ϵ of the monoid during the definition.

The generalized judgment has the form $\Delta \mid \Gamma \; ; \; \Omega \vdash e : A$ where Γ contains type assignments for variables that can be used in an unrestricted (not linear and not ordered) way. All the previous rules are augmented by propagating Γ from the conclusion to all premises. Because our term language is untyped, no extensions are needed there. Similarly, the rules of our dynamics do not need to change.

$$\frac{}{\Delta \mid \Gamma, x : A \; ; \cdot \vdash x : A}\ \mathsf{hyp}$$

$$\frac{\Delta \mid \Gamma, x : A \; ; \Omega \vdash e : B}{\Delta \mid \Gamma \; ; \Omega \vdash \lambda x.\, e : A \to B}\ {\to}I \qquad \frac{\Delta \mid \Gamma \; ; \Omega \vdash e_1 : A \to B \quad \Delta \mid \Gamma \; ; \cdot \vdash e_2 : A}{\Delta \mid \Gamma \; ; \Omega \vdash e_1\, e_2 : B}\ {\to}E$$

Figure 3 Unrestricted functions.

We extend the logical predicate using arguments not afforded any resources.

$$m \Vdash v \in [A \to B] \quad \Longleftrightarrow \quad \forall w.\, \epsilon \Vdash w \in [A] \Longrightarrow m \Vdash v\, w \in [\![B]\!]$$

The fundamental theorem extends in a straightforward way.

▶ **Theorem 6** (Fundamental Theorem (mixed ordered/unrestricted))**.** *Assume $\Delta \mid \Gamma \; ; \Omega \vdash e : A$, a mapping S with domain Δ, and two closing substitutions $\epsilon \Vdash^S \theta \in [\Gamma]$ and $m \Vdash^S \eta \in [\Omega]$. Then $m \Vdash^S (\theta \; ; \eta)(e) \in [\![A]\!]$.*

Proof. By induction on the structure of the given typing derivation. ◀

An interesting side effect of these definitions is that if we omit ordered functions but retain pairs we obtain the "usual" formulation of closed logical predicates, including certain consequences of parametricity for the ordinary λ-calculus.

▶ **Theorem 7.** *If*

$$\cdot \vdash e : \forall \alpha.\, \alpha \to (\alpha \to (\alpha \bullet \alpha))$$

then e is extensionally equivalent to one of 4 functions: $\lambda x.\, \lambda y.\, (x, y)$, $\lambda x.\, \lambda y.\, (y, x)$, $\lambda x.\, \lambda y.\, (x, x)$, or $\lambda x.\, \lambda y.\, (y, y)$.

Proof. By the fundamental theorem, we have

$$\epsilon \Vdash e \in [\![\forall \alpha.\, \alpha \to (\alpha \to (\alpha \bullet \alpha))]\!]$$

We use this for an abitrary closed type A with two arbitary values v, and w and relation R_A with $\epsilon\ R_A\ v$ and $\epsilon\ R_A\ w$. Exploiting the definition, we get

$$\epsilon \Vdash^{\alpha \mapsto R_A} e \in [\![\alpha \to (\alpha \to (\alpha \bullet \alpha))]\!]$$

Using the definition of function twice and skipping over some evaluation and reverse evaluation, we obtain

$$\epsilon \Vdash^{\alpha \mapsto R_A} e\, v\, w \in [\![\alpha \bullet \alpha]\!]$$

This means that $e\, v\, w \hookrightarrow (u_1, u_2)$ with $\epsilon\ R_A\ u_1$ and $\epsilon\ R_A\ u_2$. Because of the definition of R_A there are 4 possibilities for (u_1, u_2), namely (v, w), (w, v), (v, v) and (w, w). This in turn means e is extensionally equal to one of the 4 functions shown. ◀

6 Sums, Twist, and Recursive Types

At this point, we are at a crossroads. Because we would like to prove theorems regarding more complex data structures such as lists, trees, or streams, we could extend the development with general inductive and coinductive types and their recursors. We conjecture that this is possible and leave it to future work. The other path is to work with *purely positive types*, including recursive ones whose values can be directly observed. In this approach, the definition of the logical predicate is quite easy to extend. It becomes a nested inductive definition: either the type becomes smaller or, once we encounter a purely positive type and recursion is possible, from then on the terms become strictly smaller. In this paper we take the latter approach, which excludes coinductive types such as streams from consideration, but still yields many interesting and intuitive consequences.

We take the opportunity to also round out our language with sums and twist (the symmetric counterpart of fuse). We use a signature defining *equirecursive type names* that may be arbitrarily mutually recursive. Because such type definitions are otherwise closed, they constitute metavariables in the sense of contextual modal type theory [31]. Each type definition $F[\Delta] = A^+$ (for Δ a context of type variables as in previous sections) must be *contractive*, that is, its definiens cannot be be another type name. Moreover, A^+ must be *purely positive*, which is interpreted *inductively*.

Types	A	$::=$	$\ldots \mid A \circ B \mid \oplus\{\ell : A_\ell\}_{\ell \in L}$
Purely Positive Types	A^+, B^+	$::=$	$A^+ \bullet B^+ \mid A^+ \circ B^+ \mid \mathbf{1} \mid \oplus\{\ell : A^+_\ell\}_{\ell \in L} \mid F[\theta]$
Type Definitions	Σ	$::=$	$F[\Delta] = A^+ \mid (\cdot) \mid \Sigma_1, \Sigma_2$
Type Substitutions	θ	$::=$	$\alpha \mapsto A^+ \mid (\cdot) \mid \theta_1\, \theta_2$

The language of terms does not change much because type names are equirecursive.

$$\begin{array}{llll} \text{Term} & e & ::= & \ldots \\ & & \mid & k(e) \mid \textbf{match}\ e\ \{\ell(x_\ell) \Rightarrow e'\}_{\ell \in L} \quad (\oplus\{\ell : A_\ell\}) \end{array}$$

We add the type $A \circ B$ ("twist"), symmetric to $A \bullet B$, since encoding it as $B \bullet A$ requires rewriting terms, flipping the order of pairs. For $A \circ B$ it is merely the typechecking that changes. This allows more types to be assigned to the same term. We allow silent unfolding of type definitions, so there are no explicit rules for $F[\theta]$.

$$\frac{\Delta \mid \Gamma \,;\, \Omega_A \vdash e_1 : A \quad \Delta \mid \Gamma \,;\, \Omega_B \vdash e_2 : B}{\Delta \mid \Gamma \,;\, \Omega_B\, \Omega_A \vdash (e_1, e_2) : A \circ B} \circ I$$

$$\frac{\Delta \mid \Gamma \,;\, \Omega \vdash e : A \circ B \quad \Delta \mid \Gamma \,;\, \Omega_L\, (y : B)\, (x : A)\, \Omega_R \vdash e' : C}{\Delta \mid \Gamma \,;\, \Omega_L\, \Omega\, \Omega_R \vdash \textbf{match}\ e\ ((x, y) \Rightarrow e') : C} \circ E$$

$$\frac{(k \in L) \quad \Delta \mid \Gamma \,;\, \Omega \vdash e : A_k}{\Delta \mid \Gamma \,;\, \Omega \vdash k(e) : \oplus\{\ell : A_\ell\}_{\ell \in L}} \oplus I$$

$$\frac{\Delta \mid \Gamma \,;\, \Omega \vdash e : \oplus\{\ell : A_\ell\}_{\ell \in L} \quad (\Delta \mid \Gamma \,;\, \Omega_L\, (x_\ell : A_\ell)\, \Omega_R \vdash e_\ell : A_\ell) \quad (\forall \ell \in L)}{\Delta \mid \Gamma \,;\, \Omega_L\, \Omega\, \Omega_R \vdash \textbf{match}\ e\ \{\ell(x_\ell) \Rightarrow e_\ell\}_{\ell \in L} : C} \oplus E$$

Figure 4 Ordered Natural Deduction, Extended.

$$\frac{e \hookrightarrow v}{k(e) \hookrightarrow k(v)} \qquad \frac{e \hookrightarrow k(v) \quad [v/x_k]e_k \hookrightarrow v'}{\textbf{match}\ e\ \{\ell(x_\ell) \Rightarrow e_\ell\}_{\ell \in L} \hookrightarrow v'}$$

Figure 5 Big-Step Operational Semantics, Extended.

The logical predicate is also extended in a straightforward manner. We assume the signature Σ is fixed and therefore do not carry it explicitly through the definitions.

$$\begin{array}{lcl} m \Vdash^S v \in [A \circ B] & \iff & \exists m_1, m_2.\ m = m_2 \cdot m_1 \wedge v = (v_1, v_2) \\ & & \wedge\, m_1 \Vdash^S v_1 \in [A] \wedge m_2 \Vdash^S v_2 \in [B] \\ m \Vdash^S k(v) \in [\oplus\{\ell : A_\ell\}_{\ell \in L}] & \iff & m \Vdash^S v \in [A_k] \wedge k \in L \\ m \Vdash^S v \in [F[\theta]] & \iff & m \Vdash^S v \in \theta(A^+) \text{ where } F[\Delta] = A^+ \in \Sigma \end{array}$$

Because we have equirecursive type definitions, the last clause is usually applied silently. The definition of the logical predicate is no longer straightforwardly inductive on the structure of the type. However, we see that for purely positive types (the only ones involved in recursion), the *value* in the definition becomes strictly smaller in each clause if type definitions are contractive. In other words, we now have a nested inductive definition of the logical predicate, first on the type, and when the type is purely positive, on the structure of the value.

We can also add recursion to our term language with the key proviso that we either restrict ourselves to certain patterns of recursion (for example, primitive recursion), or termination is guaranteed by other external means (for example, an analysis using sized types [2]). This assumption allows us to maintain the structure of the logical predicate, even if it is no longer a means to prove termination (which we are not interested in for this paper).

▶ **Lemma 8** (Compositionality (including purely positive equirecursive types)). *Define* R_A *such that* $k\ R_A\ w$ *iff* $k \Vdash w \in [A]$. *Then* $m \Vdash^{S,\alpha\mapsto R_A} v \in [B(\alpha)]$ *iff* $m \Vdash^S v \in [B(A)]$.

Proof. By nested induction on the definition of the logical predicate for $B(\alpha)$, first on the structure of B and second on the structure of the value when a purely positive type $F[\theta]$ has been reached. ◀

▶ **Theorem 9** (Fundamental Theorem (including purely positive recursive types)). *Assume* $\Delta \mid \Gamma\, ;\, \Omega \vdash e : A$, *a mapping* S *with domain* Δ, *and two closing substitutions* $\epsilon \Vdash^S \theta \in [\Gamma]$ *and* $m \Vdash^S \eta \in [\Omega]$. *Then* $m \Vdash^S (\theta\, ;\, \eta)(e) \in [\![A]\!]$.

Proof. By induction on the structure of the given typing derivation. When reasoning about functions and recursion, we need the assumption of termination. ◀

7 Free Theorems for Ordered Lists

We start with some theorems about ordered lists, not unlike those analyzed by Wadler [46], but much sharper due to substructural typing. We define two versions of ordered lists, one that is ordered left-to-right and one that is ordered right-to-left. Both of these use exactly the same representation; just their typing is different.

$$\begin{aligned}
&\mathit{llist}\ \alpha = \oplus\{\underline{\mathsf{nil}} : \mathbf{1}, \underline{\mathsf{cons}} : \alpha \bullet \mathit{llist}\ \alpha\}\\
&\mathit{rlist}\ \alpha = \oplus\{\underline{\mathsf{nil}} : \mathbf{1}, \underline{\mathsf{cons}} : \alpha \circ \mathit{rlist}\ \alpha\}
\end{aligned}$$

The following will be a useful lemma about ordered lists.

▶ **Lemma 10** (Ordered Lists).

$$\begin{aligned}
m \Vdash^S v \in [\mathit{llist}\ \alpha] \quad\Longleftrightarrow\quad & m = \epsilon \wedge v = \underline{\mathsf{nil}}\,(\,)\\
& \vee\ \exists m_1, m_2.\, m = m_1 \cdot m_2 \wedge v = \underline{\mathsf{cons}}\,(v_1, v_2)\\
& \quad \wedge\ m_1\ S(\alpha)\ v_1 \wedge m_2 \Vdash v_2 \in [\mathit{llist}\ \alpha]
\end{aligned}$$

$$\begin{aligned}
m \Vdash^S v \in [\mathit{rlist}\ \alpha] \quad\Longleftrightarrow\quad & m = \epsilon \wedge v = \underline{\mathsf{nil}}\,(\,)\\
& \vee\ \exists m_1, m_2.\, m = m_2 \cdot m_1 \wedge v = \underline{\mathsf{cons}}\,(v_1, v_2)\\
& \quad \wedge\ m_1\ S(\alpha)\ v_1 \wedge m_2 \Vdash v_2 \in [\mathit{rlist}\ \alpha]
\end{aligned}$$

Proof. By unrolling the definitions of the logical predicate and the equirecursive nature of the definition of lists. ◀

For the applications, we abbreviate lists, writing $[v_1, \ldots, v_n]$ for $\underline{\mathsf{cons}}(v_1, \ldots, \underline{\mathsf{cons}}(v_n, \underline{\mathsf{nil}}\,(\,)))$.

$$m \Vdash^{\alpha\mapsto R_A} v \in [\mathit{llist}\ \alpha] \Longleftrightarrow m = m_1 \cdots m_n, v = [v_1, \ldots, v_n] \text{ where } m_i\ R_A\ v_i \text{ (for some } m_i, v_i)$$

$$m \Vdash^{\alpha\mapsto R_A} v \in [\mathit{rlist}\ \alpha] \Longleftrightarrow m = m_n \cdots m_1, v = [v_1, \ldots, v_n] \text{ where } m_i\ R_A\ v_i \text{ (for some } m_i, v_i)$$

Now we state a first property of lists that follows as a consequence of our parameterized logical predicate.

▶ **Theorem 11.** *If* $\cdot \vdash f : \forall\alpha.\, \mathit{llist}\ \alpha \twoheadrightarrow \mathit{llist}\ \alpha$ *then* f *is extensionally equal to the identity function on lists.*

Proof. By the fundamental theorem, we have

$$\epsilon \Vdash f \in [\forall\alpha.\, \mathit{llist}\ \alpha \twoheadrightarrow \mathit{llist}\ \alpha]$$

To construct a relation R_A we pick an arbitary closed type A. For the monoid, we pick the one freely generated by $a_1, a_2, \ldots$ and define

$$m\ R_A\ v \iff m = a_i \wedge v = v_i$$

for arbitrary elements v_i. By definition, we obtain, for $S = \alpha \mapsto R_A$,

$$\epsilon \Vdash^S f \in [\mathit{llist}\ \alpha \twoheadrightarrow \mathit{llist}\ \alpha]$$

Again by definition, that's the case iff

$$\forall m, v.\, m \Vdash^S v \in [\mathit{llist}\ \alpha] \Longrightarrow \epsilon \cdot m \Vdash^S f\, v \in [\![\mathit{llist}\ \alpha]\!]$$

Here, $\epsilon \cdot m = m$, by the monoid laws. Therefore $f\, v \hookrightarrow w$ and

$$\forall m, v.\, m \Vdash^S v \in [\mathit{llist}\ \alpha] \Longrightarrow m \Vdash^S w \in [\mathit{llist}\ \alpha]$$

We use this for $m = a_1 \cdots a_n$ and $v = [v_1, \ldots, v_n]$. By our lemma about lists and the arbitrary nature of A and v_i we conclude that $w = v$. ◀

By similar reasoning we can obtain the following properties.

▶ **Theorem 12.**
1. *If $f : \forall\alpha.\mathit{rlist}\ \alpha \twoheadrightarrow \mathit{rlist}\ \alpha$ then f is extensionally equal to the identity function.*
2. *If $f : \forall\alpha.\mathit{rlist}\ \alpha \twoheadrightarrow \mathit{llist}\ \alpha$ then f is extensionally equal to the list reversal function.*
3. *If $f : \forall\alpha.\mathit{llist}\ \alpha \twoheadrightarrow \mathit{rlist}\ \alpha$ then f is extensionally equal to the list reversal function.*

Proof. By very similar reasoning to the one in Theorem 11. ◀

But can we deduce properties of higher-order functions using ordered parametricity? We show one primary example; others such as *map* follow directly from it or similarly.

Unlike the usual or even linear parametricity, the type of *fold* in ordered natural deduction guarantees that it must be *the* fold function! (The essential difference being that in, say, linear natural deduction, a function having the same type as *fold* may still apply a permutation to its argument before performing the fold.) Note that the combining function and initial element are unrestricted arguments (one is called for every list element, and one is called only for the empty list), but that the combining function's arguments are ordered.

▶ **Theorem 13.** *If*

$$\cdot \vdash f : \forall\alpha.\, \forall\beta.\, (\alpha \bullet \beta \twoheadrightarrow \beta) \to \beta \to \mathit{llist}\ \alpha \twoheadrightarrow \beta$$

then f extensionally equal to the fold function, that is,

$$f\ g\ b\ [v_1, v_2, \ldots, v_n] = g(v_1, g(v_2, \ldots, g(v_n, b)))$$

Proof. We use the free monoid over constructors $a_1, a_2, \ldots$. Furthermore, given a type A with arbitrary elements v_i we define the relation R_A by

$$m\ R_A\ v \iff m = a_i \wedge v = v_i \text{ for some } i$$

Since the type involves another quantified type β, we need to define a second relation R_B where

$$m \; R_B \; d \iff m = a_{i_1} \cdots a_{i_k} \wedge d = g(v_{i_1}, g(v_{i_2}, \ldots, g(v_{i_k}, b)))$$

With these relations and the definition on of the logical predicate we get the following two properties.

1. $\forall m_1, m_2, v, d.\, m_1 \; R_A \; v \wedge m_2 \; R_B \; d \Longrightarrow m_1 \cdot m_2 \; R_B \; g(v, d)$
2. $\epsilon \; R_B \; g$

Since

$$a_1 \cdots a_n \Vdash^{\alpha \mapsto R_A} [v_1, \ldots, v_n] \in [\mathit{llist}\; \alpha]$$

we can use the second and iterate the first property to conclude that

$$a_1 \cdots a_n \; R_B \; w \qquad \text{for } f \; g \; b \; [v_1, \ldots, v_n] \hookrightarrow w$$

By definition of R_B, this yields

$$f \; g \; b \; [v_1, \ldots, v_n] = g(v_1, \ldots g(v_n, b))$$

in the sense that both sides evaluate to w. Because functions and values were chosen arbitrarily, this expresses the desired extensional equality. ◀

8 Free Theorems Regarding Trees

Consider

$$\begin{aligned}
\mathit{lxrtree}\; \alpha &= \oplus\{\underline{\mathsf{leaf}} : \mathbf{1}, \underline{\mathsf{cons}} : \mathit{lxrtree}\; \alpha \bullet \alpha \bullet \mathit{lxrtree}\; \alpha\} \\
\mathit{xlrtree}\; \alpha &= \oplus\{\underline{\mathsf{leaf}} : \mathbf{1}, \underline{\mathsf{cons}} : (\mathit{xlrtree}\; \alpha \circ \alpha) \bullet \mathit{xlrtree}\; \alpha\} \\
\mathit{lrxtree}\; \alpha &= \oplus\{\underline{\mathsf{leaf}} : \mathbf{1}, \underline{\mathsf{cons}} : \mathit{lrxtree}\; \alpha \bullet (\alpha \circ \mathit{xlrtree}\; \alpha)\}
\end{aligned}$$

Here are a few free theorems regarding such trees. Further variations exist.

▶ **Theorem 14.**

1. *If $f : \forall \alpha.\, \mathit{lxrtree}\; \alpha \twoheadrightarrow \mathit{llist}\; \alpha$ then $f\; t$ lists the elements of t following an* inorder traversal.
2. *If $f : \forall \alpha.\, \mathit{xlrtree}\; \alpha \twoheadrightarrow \mathit{llist}\; \alpha$ then $f\; t$ lists the elements of t following a* preorder traversal.
3. *If $f : \forall \alpha.\, \mathit{lrxtree}\; \alpha \twoheadrightarrow \mathit{llist}\; \alpha$ then $f\; t$ lists the elements of t following a* postorder traversal.

Proof. Trees, like lists, are purely positive types. As such, we can prove an analogue of Lemma 10. We only show one of them, writing t for tree values.

$$\begin{aligned}
m \Vdash^S t \in [\mathit{lxrtree}\; \alpha] \quad \iff \quad & m = \epsilon \wedge t = \underline{\mathsf{leaf}}(\,) \\
& \vee \exists m_1, k, m_2.\, m = m_1 \cdot k \cdot m_2 \wedge v = \underline{\mathsf{node}}(t_1, v, t_2) \\
& \quad \wedge m_1 \Vdash^S t_1 \in [\mathit{lxrtree}\; \alpha] \wedge k \; S(\alpha) \; v \wedge m_2 \Vdash^S t_2 \in [\mathit{lxrtree}\; \alpha]
\end{aligned}$$

◀

9 From Ordered to Linear Types

Exploring parametricity for *linear* types instead of ordered ones is now a rather straightforward change. We conflate the left and right implication into a single implication, and similarly for conjunction.

ordered	linear	structural	values
$B \twoheadrightarrow A$	$A \multimap B$	$A \rightarrow B$	$\lambda x.\, e$
$A \rightarrowtail B$			
$A \bullet B$	$A \otimes B$	$A \times B$	(v_1, v_2)
$A \circ B$			
$\mathbf{1}$	$\mathbf{1}$	$\mathbf{1}$	$(\,)$
$\oplus\{\ell : A_\ell\}$	$\oplus\{\ell : A_\ell\}$	$\oplus\{\ell : A_\ell\}$	$\ell(v)$

We see that in the transition from the linear to the structural case, no further connectives collapse. That's because we would still distinguish eager pairs ($A \times B$) from lazy records that we have elided from our development since they do not introduce any fundamentally new ideas.

From the point of view of typing, the easiest change is to just permit the silent rule of exchange

$$\frac{\Delta \mid \Gamma \,;\, \Omega_L \, (y : B) \, (x : A) \, \Omega_R \vdash e : C}{\Delta \mid \Gamma \,;\, \Omega_L \, (x : A) \, (y : B) \, \Omega_R \vdash e : C} \;\textsf{exchange}$$

The more typical change is to replace context concatenation $\Omega_L \, \Omega_R$ with context merge $\Omega_L \bowtie \Omega_R$ which allows arbitrary interleavings of the hypotheses.

Our definition of the logical predicates remains that same, except that we assume that the algebraic structure parameterizing our definitions is a *commutative monoid*. This immediately validates the rules of exchange and the fundamental theorem goes through as before.

The results of exploiting the fundamental theorem to obtain parametricity results are no longer as sharp. For example:

▶ **Theorem 15.** *If* $\cdot \vdash e : \forall \alpha.\, \alpha \multimap \alpha \multimap \alpha \otimes \alpha$ *then* f *is extensionally equal to* $\lambda x.\, \lambda y.\, (x, y)$ *or* $\lambda x.\, \lambda y.\, (y, x)$.

Proof. By the fundamental theorem, we have

$$\epsilon \Vdash e \in [\![\forall \alpha.\, \alpha \multimap \alpha \multimap \alpha \otimes \alpha]\!]$$

Therefore $e \hookrightarrow f$ and

$$\epsilon \Vdash f \in [\forall \alpha.\, \alpha \multimap \alpha \multimap \alpha \otimes \alpha]$$

We use a free commutative monoid with two generators, a and b, arbitrary values v and w such that $a \; R \; v$ and $b \; R \; w$. By the fundamental theorem:

$$\epsilon \Vdash^{\alpha \mapsto R} f \in [\alpha \multimap \alpha \multimap \alpha \otimes \alpha]$$

Applying this function to v and w, we obtain that $f \, v \, w \hookrightarrow p$ and

$$a \cdot b \Vdash^{\alpha \mapsto R} p \in [\alpha \otimes \alpha]$$

This is true, again by definition, if for some m and k and p_1 and p_2 we have

$$m \cdot k = a \cdot b \wedge p = (p_1, p_2) \wedge m \Vdash^{\alpha \mapsto R} p_1 \in [\alpha] \wedge k \Vdash^{\alpha \mapsto R} p_2 \in [\alpha]$$

Further applying definitions, we get that for some m, k, p_1, and p_2, we have

$$m \cdot k = a \cdot b \wedge m \; R \; p_1 \wedge k \; R \; p_2$$

There are 4 ways that $a \cdot b$ could be decomposed into $m \cdot k$, but the definition of R leaves only two possibilities: $m = a$, $k = b$, $p_1 = v$ and $p_2 = w$ or $m = b$, $k = a$, $p_1 = w$ and $p_2 = v$. Summarizing: either

$$e\, v\, w \hookrightarrow (v, w)$$

or

$$e\, v\, w \hookrightarrow (w, v)$$

which expresses that e is extensionally equal to $\lambda x.\, \lambda y.\, (x, y)$ or $\lambda x.\, \lambda y.\, (y, x)$. ◀

▶ **Theorem 16.** *If* $\cdot \vdash e : \forall \alpha. \mathit{list}\, \alpha \multimap \mathit{list}\, \alpha$ *then* e *is extensionally equal to a permutation of the list elements.*

Proof. As in the proof of the related ordered theorem, we apply the fundamental theorem and then the definition for arbitrary values v_i with $a_i\ R\ v_i$ where $\alpha \mapsto R$, and the commutative monoid is freely generated from $a_1, a_2, \ldots$.

Taking analogous steps to the ordered case, we conclude that $a_1 \cdots a_n = m_1 \cdots m_n$ modulo commutativity (and associativity, as always) where each m_i is a unique a_j. ◀

In the unrestricted case where various algebraic elements are fixed to be ϵ, we can only obtain that every element of the output list must be a member of the input list, because those elements are in $\epsilon\ R\ v_i$. We do not write out the details of this straightforward adaptation of foregoing proofs.

10 Related Work

The most directly related work is Zhao et al.'s [47] open logical relation for parametricity for a dual intuitionistic-linear polymorphic lambda calculus. In this work, they define an *open* logical relation that includes an analog of typing contexts in the semantic model. While our development follows a similar structure, our resource algebraic account allows us to eliminate spurious typechecking premises in definitions and permits a more flexible range of substructural type systems.

Ahmed, Fluet, and Morrisett [4] introduce a logical relation for substructural state via step-indexing, followed by [5] a *linear language with locations* (L3) defined by a Kripke-style logical relation to account for a language with mutable storage. However, the underlying languages in these developments do not support parametric polymorphism. Ahmed, Dreyer, and Rossberg later provide a logical relations account of a System F-based language supporting imperative state update, and they demonstrate representation independence results for this system [3]. The languages modeled in this body of work represent a specific point in the design space with respect to imperative state update and references, as opposed to our more general schema for substructural types in a functional setting. However, Kripke-style logical relations that model a store as a partial commutative monoid have some parallels to our development, and drawing out a more precise relationship between these systems represents an interesting path of future work.

There are a few developments that start from different settings but develop semantics with similar properties. Pérez et al. develop logical relations for linear session types [33, 34] to establish normalization results, but there is no account of parametricity. Caires et al. [11], Derakhshan et al. [13], and Balzer et al. [8] account for parametricity in linear session-typed communication with the goal of reasoning abstractly over protocol implementations. Their logical relations do not directly capture resource usage nor obviously admit free theorems similar to ours, but rather form a basis for information flow reasoning as an orthogonal application of parametricity [14, 44].

The Iris system for program reasoning via higher-order separation logic incorporates a semantic model initially based on monoids [21], which is later extended to more general resource algebras [20]. Their parameterization over resource algebras seems to work similarly to ours, but towards the goal of program verification rather than type-based reasoning. Birkedal et al. [10] demonstrate free theorems for separation logic specifications in service of enforcing communication protocols between clients and libraries.

The use of "resource semantics" more generally to account for the semantics of substructural logics extends at least to Kamide [23] and the logic of bunched implications [32], and similar ideas have recently gained traction in the context of graded modal type systems [45]. For instance, Atkey and Wood [6] introduce a notion of "context constrained computation" to generalize parametricity reasoning over linear list operations to arbitrary semiring-graded modalities. They do not directly account for polymorphism in the type system, but can instantiate their calculus with example-specific world definitions to prove e.g. that a linear function on lists of a generic key type must return a permutation. Abel and Bernardy [1] extend this idea to a similar one that includes polymorphic types and presents free theorems. The ringoid structure that generalizes the graded approach does not, however, accommodate ordered logic.

11 Conclusion

We have provided an account of substructural parametricity including ordered, linear, and unrestricted disciplines. The fewer structural properties are supported, the more precise the characterization of a function's behavior from its type. We have also implemented an ordered type checker using a bidirectional type system with so-called additive contexts [7], but the details are beyond the scope of this paper. Suffice it to say that all the functions such as append, reverse, tree traversals, and fold can actually be implemented in a variety of ways and our "free theorems" are therefore not vacuous.

The most immediate item of future work is to support general inductive and coinductive types instead of purely positive recursive types. This would allow a new class of applications, including (productive) stream processing and object-oriented program patterns.

We also envision an adjoint combination of different substructural type systems [19], extended to include exchange among the explicit structural rules, with the logical predicates given herein properly extended to account for the different modalities present in the adjoint type system.

References

1. Andreas Abel and Jean-Philippe Bernardy. A unified view of modalities in type systems. *Proceedings of the ACM on Programming Languages*, 4(ICFP):1–28, 2020. `doi:10.1145/3408972`.
2. Andreas Abel and Brigitte Pientka. Well-founded recursion with copatterns and sized types. *Journal of Functional Programming*, 26:e2, 2016. `doi:10.1017/S0956796816000022`.
3. Amal Ahmed, Derek Dreyer, and Andreas Rossberg. State-dependent representation independence. In *36th Symposium on Principles of Programming Languages (POPL 2009)*, pages 340–353, Savannah, Georgia, USA, January 2009. ACM. `doi:10.1145/1480881.1480925`.

4 Amal Ahmed, Matthew Fluet, and Greg Morrisett. A step-indexed model of substructural state. In *10th International Conference on Functional Programming (ICFP 2005)*, pages 78–91, 2005. doi:10.1145/1086365.1086376.

5 Amal Ahmed, Matthew Fluet, and Greg Morrisett. L^3: a linear language with locations. *Fundamenta Informaticae*, 77(4):397–449, 2007.

6 Bob Atkey and James Wood. Context constrained computation. In *3rd Workshop on Type-Driven Development (TyDe'18)*, 2018.

7 Robert Atkey. Syntax and semantics of quantitative type theory. In Anuj Dawar and Erich Grädel, editors, *33rd Conference on Logic in Computer Science (LICS 2018)*, pages 56–65, Oxford, UK, July 2018. ACM. doi:10.1145/3209108.3209189.

8 Stephanie Balzer, Farzaneh Derakhshan, Robert Harper, and Yue Yao. Logical relations for session-typed concurrency. *arXiv preprint arXiv:2309.00192*, 2023. doi:10.48550/arXiv.2309.00192.

9 P. N. Benton. A mixed linear and non-linear logic: Proofs, terms and models. In Leszek Pacholski and Jerzy Tiuryn, editors, *Selected Papers from the 8th International Workshop on Computer Science Logic (CSL'94)*, pages 121–135, Kazimierz, Poland, September 1994. Springer LNCS 933. An extended version appears as Technical Report UCAM-CL-TR-352, University of Cambridge.

10 Lars Birkedal, Thomas Dinsdale-Young, Armaël Guéneau, Guilhem Jaber, Kasper Svendsen, and Nikos Tzevelekos. Theorems for free from separation logic specifications. *Proceedings of the ACM on Programming Languages*, 5(ICFP):1–29, 2021. doi:10.1145/3473586.

11 Luís Caires, Jorge A Pérez, Frank Pfenning, and Bernardo Toninho. Behavioral polymorphism and parametricity in session-based communication. In *22nd European Symposium on Programming (ESOP 2013)*, pages 330–349, Rome, Italy, March 2013. Springer LNCS 7792. doi:10.1007/978-3-642-37036-6_19.

12 Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In P.Gastin and F.Laroussinie, editors, *21st International Conference on Concurrency Theory (CONCUR 2010)*, pages 222–236, Paris, France, August 2010. Springer LNCS 6269. doi:10.1007/978-3-642-15375-4_16.

13 Farzaneh Derakhshan, Stephanie Balzer, and Limin Jia. Session logical relations for noninterference. In *36th Annual Symposium on Logic in Computer Science (LICS 2021)*, pages 1–14. IEEE, 2021. doi:10.1109/LICS52264.2021.9470654.

14 Farzaneh Derakhshan, Stephanie Balzer, and Yue Yao. Regrading policies for flexible information flow control in session-typed concurrency (artifact). *Dagstuhl Artifacts Series*, 10(2):4–1, 2024.

15 Peng Fu, Kohei Kishida, and Peter Selinger. Linear dependent type theory for quantum programming languages. In *35th Symposium on Logic in Computer Science (LICS 2020)*, pages 440–453, Saarbrücken, Germany, July 2020. ACM.

16 Simon J. Gay and Vasco T. Vasconcelos. Linear type theory for asynchronous session types. *Journal of Functional Programming*, 20(1):19–50, January 2010. doi:10.1017/S0956796809990268.

17 Aïna Linn Georges, Benjamin Peters, Laila Albeheiry, Leo White, Stephan Dolan, Richard A. Eisenberg, Chris Casinghino, François Pottier, and Derek Dreyer. Data race freedom à la mode. In *Principles of Programming Languages (POPL 2025)*, volume 9 of *Proceedings on Programming Languages*, pages 656–686. ACM, January 2025. doi:10.1145/3704859.

18 Jean-Yves Girard and Yves Lafont. Linear logic and lazy computation. In H. Ehrig, R. Kowalski, G. Levi, and U. Montanari, editors, *Proceedings of the International Joint Conference on Theory and Practice of Software Development*, volume 2, pages 52–66, Pisa, Italy, March 1987. Springer-Verlag LNCS 250. doi:10.1007/BFB0014972.

19 Junyoung Jang, Sophia Roshal, Frank Pfenning, and Brigitte Pientka. Adjoint natural deduction. In Jakob Rehof, editor, *9th International Conference on Formal Structures for Computation and Deduction (FSCD 2024)*, pages 15:1–15:23, Tallinn, Estonia, July 2024. LIPIcs 299. Extended version available as https://arxiv.org/abs/2402.01428.

20 Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28:e20, 2018. doi:10.1017/S0956796818000151.

21 Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *42nd Annual Symposium on Principles of Programming Languages (POPL 2015)*, pages 637–650, Mumbai, India, January 2015. ACM. doi:10.1145/2676726.2676980.

22 Gilles Kahn. Natural semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, pages 22–39. Springer-Verlag LNCS 247, 1987. doi:10.1007/BFB0039592.

23 Norihiro Kamide. Kripke semantics for modal substructural logics. *Journal of Logic, Language and Information*, 11:453–470, 2002. doi:10.1023/A:1019915908844.

24 Max Kanovich, Stepan Kuznetsov, Vivek Nigam, and Andre Scedrov. A logical framework with commutative and non-commutative subexponentials. In *International Joint Conference on Automated Reasoning (IJCAR 2018)*, pages 228–245. Springer LNAI 10900, 2018. doi:10.1007/978-3-319-94205-6_16.

25 Joachim Lambek. The mathematics of sentence structure. *The American Mathematical Monthly*, 65(3):154–170, 1958.

26 Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying Rust programs using linear ghost types. In *Proceedings of the ACM on Programming Languages*, volume 7 (OOPSLA 2023), pages 286–315, April 2023. Extended version available as https://arxiv.org/abs/2303.05491.

27 Anton Lorenzen, Daan Leijen, and Wouter Swierstra. FP^2: Fully in-place functional programming. In *International Conference on Functional Programming (ICFP 2023)*, Proceedings on Programming Languages, pages 275–304. ACM, August 2023. doi:10.1145/3607840.

28 Anton Lorenzen, Daan Leijen, Wouter Swierstra, and Sam Lindley. The functional essence of imperative binary search trees. In *Programming Language Design and Implementation (PLDI 2024)*, volume 8 of *Proceedings on Programming Languages*, pages 518–542. ACM, January 2024. doi:10.1145/3656398.

29 Wendy MacCaull. Relational semantics and a relational proof system for full Lambek calculus. *Journal of Symbolic Logic*, 63(2):623–637, 1998. doi:10.2307/2586855.

30 John C Mitchell. Representation independence and data abstraction. In *Proceedings of the 13th Symposium on Principles of Programming Languages (POPL 1986)*, pages 263–276. ACM, 1986. doi:10.1145/512644.512669.

31 Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *Transactions on Computational Logic*, 9(3), 2008. doi:10.1145/1352582.1352591.

32 Peter W O'Hearn and David J Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, 1999. doi:10.2307/421090.

33 Jorge A. Pérez, Luís Caires, Frank Pfenning, and Bernardo Toninho. Termination in session-based concurrency via linear logical relations. In H. Seidl, editor, *22nd European Symposium on Programming (ESOP 2012)*, pages 539–558, Tallinn, Estonia, March 2012. Springer LNCS 7211.

34 Jorge A. Pérez, Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logical relations and observational equivalences for session-based concurrency. *Information and Computation*, 239:254–302, 2014. doi:10.1016/J.IC.2014.08.001.

35 Frank Pfenning and Klaas Pruiksma. Relating message passing and shared memory, proof-theoretically. In S. Jongmans and A. Lopes, editors, *25th International Conference on Coordination Models and Languages (COORDINATION 2023)*, pages 3–27, Lisbon, Portugal, June 2023. Springer LNCS 13908. Notes to an invited talk. doi:10.1007/978-3-031-35361-1_1.

36 Jeff Polakow. *Ordered Linear Logic and Applications*. PhD thesis, Department of Computer Science, Carnegie Mellon University, August 2001.

37 Jeff Polakow and Frank Pfenning. Natural deduction for intuitionistic non-commutative linear logic. In J.-Y. Girard, editor, *Proceedings of the 4th International Conference on Typed Lambda Calculi and Applications (TLCA'99)*, pages 295–309, L'Aquila, Italy, April 1999. Springer-Verlag LNCS 1581. `doi:10.1007/3-540-48959-2_21`.

38 Klaas Pruiksma, Willow Chargin, Frank Pfenning, and Jason Reed. Adjoint logic and its concurrent operational interpretation. Unpublished manuscript, January 2018.

39 Jason Reed. Hybridizing a logical framework. In *Proceedings of the International Workshop on Hybrid Logic (HyLo'06)*, pages 135–148. Electronic Notes in Theoretical Computer Science, v.174(6), 2007.

40 Jason Reed and Frank Pfenning. A constructive approach to the resource semantics of substructural logics. Unpublished Manuscript, May 2010. URL: `https://www.cs.cmu.edu/~jcreed/papers/rp-substruct.pdf`.

41 Jason C. Reed. *A Hybrid Logical Framework*. PhD thesis, Carnegie Mellon University, September 2009. Available as Technical Report CMU-CS-09-155.

42 John C. Reynolds. Types, abstraction, and parametric polymorphism. In R.E.A. Mason, editor, *Information Processing 83*, pages 513–523. Elsevier, September 1983.

43 Christopher Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13:11–49, 2000. Notes for lecture course given at the International Summer School in Computer Programming at Copenhagen, Denmark, August 1967. `doi:10.1023/A:1010000313106`.

44 Bas van den Heuvel, Farzaneh Derakhshan, and Stephanie Balzer. Information flow control in cyclic process networks. In *38th European Conference on Object-Oriented Programming (ECOOP 2024)*, pages 40:1–40:30. LIPIcs 313, 2024. `doi:10.4230/LIPICS.ECOOP.2024.40`.

45 Victoria Vollmer, Danielle Marshall, Harley Eades, III, and Dominic Orchard. A mixed linear and graded logic: Proofs, terms, and models. In *33rd Conference on Computer Science Logic (CSL 2025)*, pages 32:1–32:21, Amsterdam, The Netherlands, February 2025. LIPIcs 326. `doi:10.4230/LIPICS.CSL.2025.32`.

46 Philip Wadler. Theorems for free! In J. Stoy, editor, *Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture (FPCA'89)*, pages 347–359, London, UK, September 1989. ACM. `doi:10.1145/99370.99404`.

47 Jianzhou Zhao, Qi Zhang, and Steve Zdancewic. Relational parametricity for a polymorphic linear lambda calculus. In K. Ueda, editor, *8th Asian Symposium on Programming Languages and Systems (APLAS 2010)*, pages 344–359. Springer LNCS 6461, 2010. `doi:10.1007/978-3-642-17164-2_24`.

A Complete definition of the Type System

$$\begin{array}{llll} \text{Types} & A, B & ::= & \alpha \mid 1 \mid A \bullet B \mid A \rightarrowtail B \mid A \twoheadrightarrow B \mid \forall \alpha.\, A \\ & & \mid & A \to B \mid A \circ B \mid \oplus\{\ell : A_\ell\}_{\ell \in L} \end{array}$$

$$\begin{array}{llll} \text{Purely Positive Types} & A^+, B^+ & ::= & A^+ \bullet B^+ \mid A^+ \circ B^+ \\ & & \mid & \mathbf{1} \mid \oplus\{\ell : A^+_\ell\}_{\ell \in L} \mid F[\theta] \end{array}$$

$$\text{Type Definitions} \quad \Sigma \quad ::= \quad F[\Delta] = A^+ \mid (\cdot) \mid \Sigma_1, \Sigma_2$$

$$\text{Type Substitutions} \quad \theta \quad ::= \quad \alpha \mapsto A^+ \mid (\cdot) \mid \theta_1 \theta_2$$

$$\begin{array}{lllll} \text{Terms} & e & ::= & x & \\ & & \mid & () \mid \mathbf{match}\ e\ (() \Rightarrow e') & (\mathbf{1}) \\ & & \mid & (e_1, e_2) \mid \mathbf{match}\ e\ ((x, y) \Rightarrow e') & (A \bullet B) \\ & & \mid & \lambda x.\ e \mid e_1\, e_2 & (A \rightarrowtail B, A \twoheadrightarrow B, A \to B) \\ & & \mid & k(e) \mid \mathbf{match}\ e\ \{\ell(x_\ell) \Rightarrow e'\}_{\ell \in L} & (\oplus\{\ell : A_\ell\}) \end{array}$$

Typing rules:

$$\frac{}{\Delta \mid \Gamma \,;\, x : A \vdash x : A}\ \mathsf{hyp}$$

$$\frac{}{\Delta \mid \Gamma \,;\, \cdot \vdash () : \mathbf{1}}\ \mathbf{1}I \qquad \frac{\Delta \mid \Gamma \,;\, \Omega \vdash e : \mathbf{1} \quad \Delta \mid \Gamma \,;\, \Omega_L \Omega_R \vdash e' : C}{\Delta \mid \Gamma \,;\, \Omega_L \Omega \Omega_R \vdash \mathbf{match}\ e\ (() \Rightarrow e') : A}\ \mathbf{1}E$$

$$\frac{\Delta \mid \Gamma \,;\, \Omega\,(x : A) \vdash e : B}{\Delta \mid \Gamma \,;\, \Omega \vdash \lambda x.\, e : A \twoheadrightarrow B}\ {\twoheadrightarrow}I \qquad \frac{\Delta \mid \Gamma \,;\, \Omega \vdash e_1 : A \twoheadrightarrow B \quad \Delta \mid \Gamma \,;\, \Omega_A \vdash e_2 : A}{\Delta \mid \Gamma \,;\, \Omega\, \Omega_A \vdash e_1\, e_2 : B}\ {\twoheadrightarrow}E$$

$$\frac{\Delta \mid \Gamma \,;\, (x : A)\, \Omega \vdash e : B}{\Delta \mid \Gamma \,;\, \Omega \vdash \lambda x.\, e : A \rightarrowtail B}\ {\rightarrowtail}I \qquad \frac{\Delta \mid \Gamma \,;\, \Omega \vdash e_1 : A \rightarrowtail B \quad \Delta \mid \Gamma \,;\, \Omega_A \vdash e_2 : A}{\Delta \mid \Gamma \,;\, \Omega_A\, \Omega \vdash e_1\, e_2 : B}\ {\rightarrowtail}E$$

$$\frac{\Delta \mid \Gamma \,;\, \Omega_A \vdash e_1 : A \quad \Delta \mid \Gamma \,;\, \Omega_B \vdash e_2 : B}{\Delta \mid \Gamma \,;\, \Omega_A\, \Omega_B \vdash (e_1, e_2) : A \bullet B}\ {\bullet}I$$

$$\frac{\Delta \mid \Gamma \,;\, \Omega \vdash e : A \bullet B \quad \Delta \mid \Gamma \,;\, \Omega_L\,(x : A)\,(y : B)\,\Omega_R \vdash e' : C}{\Delta \mid \Gamma \,;\, \Omega_L\, \Omega\, \Omega_R \vdash \mathbf{match}\ e\ ((x, y) \Rightarrow e') : C}\ {\bullet}E$$

$$\frac{\Delta, \alpha\ \mathsf{type} \mid \Gamma \,;\, \Omega \vdash e : A}{\Delta \mid \Gamma \,;\, \Omega \vdash e : \forall \alpha.\, A}\ \forall I \qquad \frac{\Delta \mid \Gamma \,;\, \Omega \vdash e : \forall \alpha. A(\alpha) \quad \Delta \vdash B\ \mathsf{type}}{\Delta \mid \Gamma \,;\, \Omega \vdash e : A(B)}\ \forall E$$

$$\frac{}{\Delta \mid \Gamma, x : A \,;\, \cdot \vdash x : A}\ \mathsf{hyp}$$

$$\frac{\Delta \mid \Gamma, x : A \,;\, \Omega \vdash e : B}{\Delta \mid \Gamma \,;\, \Omega \vdash \lambda x.\, e : A \to B}\ {\to}I \qquad \frac{\Delta \mid \Gamma \,;\, \Omega \vdash e_1 : A \to B \quad \Delta \mid \Gamma \,;\, \cdot \vdash e_2 : A}{\Delta \mid \Gamma \,;\, \Omega \vdash e_1\, e_2 : B}\ {\to}E$$

$$\frac{\Delta \mid \Gamma \,;\, \Omega_A \vdash e_1 : A \quad \Delta \mid \Gamma \,;\, \Omega_B \vdash e_2 : B}{\Delta \mid \Gamma \,;\, \Omega_B\, \Omega_A \vdash (e_1, e_2) : A \circ B}\ {\circ}I$$

$$\frac{\Delta \mid \Gamma \,;\, \Omega \vdash e : A \circ B \quad \Delta \mid \Gamma \,;\, \Omega_L\,(y : B)\,(x : A)\,\Omega_R \vdash e' : C}{\Delta \mid \Gamma \,;\, \Omega_L\, \Omega\, \Omega_R \vdash \mathbf{match}\ e\ ((x, y) \Rightarrow e') : C}\ {\circ}E$$

$$\frac{(k \in L) \quad \Delta \mid \Gamma \,;\, \Omega \vdash e : A_k}{\Delta \mid \Gamma \,;\, \Omega \vdash k(e) : \oplus\{\ell : A_\ell\}_{\ell \in L}}\ \oplus I$$

$$\frac{\Delta \mid \Gamma \,;\, \Omega \vdash e : \oplus\{\ell : A_\ell\}_{\ell \in L} \quad (\Delta \mid \Gamma \,;\, \Omega_L\,(x_\ell : A_\ell)\,\Omega_R \vdash e_\ell : A_\ell) \quad (\forall \ell \in L)}{\Delta \mid \Gamma \,;\, \Omega_L\, \Omega\, \Omega_R \vdash \mathbf{match}\ e\ \{\ell(x_\ell) \Rightarrow e_\ell\}_{\ell \in L} : C}\ \oplus E$$

Operational semantics:

$$\frac{}{() \hookrightarrow ()} \qquad \frac{e \hookrightarrow () \quad e' \hookrightarrow v}{\textbf{match}\ e\ (() \Rightarrow e') \hookrightarrow v}$$

$$\frac{}{\lambda x.\, e \hookrightarrow \lambda x.\, e} \qquad \frac{e_1 \hookrightarrow \lambda x.\, e_1' \quad e_2 \hookrightarrow v_2 \quad [v_2/x]e_1' \hookrightarrow v}{e_1\, e_2 \hookrightarrow v}$$

$$\frac{e_1 \hookrightarrow v_1 \quad e_2 \hookrightarrow v_2}{(e_1, e_2) \hookrightarrow (v_1, v_2)} \qquad \frac{e \hookrightarrow (v_1, v_2) \quad [v_1/x, v_2/y]e' \hookrightarrow v'}{\textbf{match}\ e\ ((x, y) \Rightarrow e') \hookrightarrow v'}$$

$$\frac{e \hookrightarrow v}{k(e) \hookrightarrow k(v)} \qquad \frac{e \hookrightarrow k(v) \quad [v/x_k]e_k \hookrightarrow v'}{\textbf{match}\ e\ \{\ell(x_\ell) \Rightarrow e_\ell\}_{\ell \in L} \hookrightarrow v'}$$

B Complete definition of the Logical Predicate

$$m \Vdash^S e \in [\![A]\!] \iff \exists v.\, e \hookrightarrow v \wedge m \Vdash^S v \in [A]$$

$$m \Vdash^S v \in [A \bullet B] \iff \exists m_1, m_2.\, m = m_1 \cdot m_2 \wedge v = (v_1, v_2) \wedge m_1 \Vdash^S v_1 \in [A] \wedge m_2 \Vdash^S v_2 \in [B]$$

$$m \Vdash^S v \in [A \twoheadrightarrow B] \iff \forall k.\, k \Vdash^S w \in [A] \Longrightarrow m \cdot k \Vdash^S v\, w \in [\![B]\!]$$

$$m \Vdash^S v \in [A \rightarrowtail B] \iff \forall k.\, k \Vdash^S w \in [A] \Longrightarrow k \cdot m \Vdash^S v\, w \in [\![B]\!]$$

$$m \Vdash^S v \in [\alpha] \iff m\ S(\alpha)\ v$$

$$m \Vdash^S v \in [\forall \alpha.\, A(\alpha)] \iff \forall B, R_B.\, m \Vdash^{S, \alpha \mapsto R_B} v \in [A(\alpha)]$$

$$m \Vdash v \in [A \to B] \iff \forall w.\, \epsilon \Vdash w \in [A] \Longrightarrow m \Vdash v\, w \in [\![B]\!]$$

$$m \Vdash^S v \in [A \circ B] \iff \exists m_1, m_2.\, m = m_2 \cdot m_1 \wedge v = (v_1, v_2) \wedge m_1 \Vdash^S v_1 \in [A] \wedge m_2 \Vdash^S v_2 \in [B]$$

$$m \Vdash^S k(v) \in [\oplus\{\ell : A_\ell\}_{\ell \in L}] \iff m \Vdash^S v \in [A_k] \wedge k \in L$$

$$m \Vdash^S v \in [F[\theta]] \iff m \Vdash^S v \in \theta(A^+) \text{ where } F[\Delta] = A^+ \in \Sigma$$