Technical Report http://hdl.handle.net/2142/107794, July 2020

Matching Logic Explained

Xiaohong Chen[1], Dorel Lucanu[2], and Grigore Roşu[1]

[1]University of Illinois at Urbana-Champaign, Champaign, USA
[2]Alexandru Ioan Cuza University, Iaşi, Romania
xc3@illinois, dlucanu@info.uaic.ro, grosu@illinois.edu

July 28, 2020

Abstract

Matching logic was recently proposed as a unifying logic for specifying and reasoning about static structure and dynamic behavior of programs. In matching logic, *patterns* and *specifications* are used to uniformly represent mathematical domains (such as numbers and Boolean values), datatypes, and transition systems, whose properties can be reasoned about using one *fixed* matching logic proof system. In this paper we give a tutorial to matching logic. We use a suite of examples to explain the basic concepts of matching logic and show how to capture many important mathematical domains, datatypes, and transition systems using patterns and specifications. We put special emphasis on the general principles of induction and coinduction in matching logic and show how to do inductive and coinductive reasoning about datatypes and codatatypes. To encourage the development of the future tools for matching logic, we propose and use throughout the paper a human-readable formal syntax to write specifications in a modular and compact way.

Keywords— matching logic, program logics, (co)inductive data types, dependent types, specification of transition systems, (co)monad specification

1 Introduction

Matching logic is a unifying logic for specifying and reasoning about static structure and dynamic behavior of programs. It was recently proposed in [1] and further developed in [2, 3]. There exist several equivalent variants of matching logic. In this paper we consider the variant that has a minimal presentation, called the *applicative matching logic*. For simplicity, we will refer to this variant simply as *matching logic* and abbreviate it as ML.

The key concept of ML is its *patterns*, which are formulas built from variables, constant symbols, one binary construct called *application*, the standard FOL constructs $\neg$, $\wedge$, $\exists$, and a least fixpoint construct μ. Semantically, patterns are interpreted as sets of elements that *match* them, which gives ML a *pattern matching semantics*. For example, $\mathbb{0}$ is a pattern matched by the natural number 0; $\mathbb{1}$ is a pattern matched by 1; $\mathbb{0} \vee \mathbb{1}$ is a (disjunctive) pattern matched by 0 and 1, or, to put it another way, an element a matches $\mathbb{0} \vee \mathbb{1}$ iff a matches $\mathbb{0}$ *or* a matches $\mathbb{1}$. Complex patterns can be built this way to match elements that are of particular structure, have certain dynamic behavior, or satisfy certain logic constraints. We discuss examples in Sections 3-9.

Patterns *constrain models*, by enforcing them to match a set of given patterns, called *axioms*. This set of axioms yields a *specification*. In this paper we will define a variety of specifications, some of them capturing relevant mathematical domains, others datatypes, and others capturing transition systems. We will also show how to build a complex specification in a modular way, by importing existing simpler specifications. To present ML specifications rigorously and compactly, we propose a *specification syntax* in Section 3 that allows us to write specifications in a compact and human-readable way. All ML specifications presented in this paper are written using this syntax.

Our main technical contribution is a collection of complete ML specifications of important datatypes and data structures (including parameterized types, function types, and dependent types), a basic process algebra and its dynamic reduction relation, and the higher-order reasoning about functors and monads in category theory. For each specification, we derive several nontrivial properties using the matching logic proof system; some of these properties require inductive/coinductive reasoning, also supported by ML.

We organize the rest of the paper as follows:

- In Section 2 we define the syntax and semantics of ML;
- In Section 3 we introduce the *specification syntax* and define the specifications of several important mathematical instruments such as equality, membership, sorts, and functions;
- In Section 4 we review the Hilbert-style proof system of ML and its soundness theorem;
- In Section 5 we explain how patterns are interpreted in ML models;
- In Section 6 we discuss the general principle of induction and coinduction in ML and compare it with the classical principle of (co)induction in complete lattices;
- In Section 7 we give specifications for examples of main data types used in programming languages: simple datatypes (booleans and naturals), parametric types (product, sum, functions, lists, and streams), dependent types (vectors, dependent product, and dependent sum). For each example we present and prove illustrative (co)inductive properties;
- In Section 8 we define a basic process algebra in ML;
- In Section 9 we use ML for higher-order reasoning in category theory and define functors, monads, and comonads as ML specifications;
- In Section 10 we conclude the paper.

2 Matching Logic Syntax and Semantics

We introduce the syntax and semantics of matching logic (ML). We refer the reader to [1, 2, 5, 6] for full technical details.

2.1 Matching Logic Syntax

ML is an unsorted logic whose formulas, called *patterns*, are built with variables, constant symbols, a binary construct called *application*, the standard FOL constructs $\bot$, $\to$, $\exists$, and a least fixpoint construct μ.

Definition 2.1. A matching logic *signature* $\mathbb{\Sigma} = (EV, SV, \Sigma)$ contains a set EV of *element variables* denoted $x, y, \ldots$, a set SV of *set variables* denoted $X, Y, \ldots$, and a set Σ of *constant symbols* (or simply *symbols*) denoted $\sigma, \sigma_1, \sigma_2, \ldots$. We require that EV and SV are countably infinite sets.

Definition 2.2. Given $\mathbb{\Sigma} = (EV, SV, \Sigma)$, the set $\textsc{Pattern}(\mathbb{\Sigma})$ of $\mathbb{\Sigma}$-*patterns* (or simply *patterns*) is inductive defined by the following grammar:

$$\varphi ::= x \mid X \mid \sigma \mid \varphi_1\, \varphi_2 \mid \bot \mid \varphi_1 \to \varphi_2 \mid \exists x.\, \varphi \mid \mu X.\, \varphi$$

where in $\mu X.\, \varphi$ we require that φ is positive in X; that is, X is not nested in an odd number of times on the left-hand side of an implication $\varphi_1 \to \varphi_2$.

We assume that application $\varphi_1\, \varphi_2$ binds the tightest and is left-associative. Both $\exists$ and μ are binders. While $\exists$ only binds element variables, μ only binds set variables. The scope of binders goes as far as possible to the right. We assume the standard notions of free variables, α-equivalence, and capture-avoiding substitution. Specifically, we use $FV(\varphi) \subseteq EV \sup SV$ to denote the set of (element and set) variables that are free in φ. We regard α-equivalent patterns as syntactically identical. We write $\varphi[\psi/x]$ (resp. $\varphi[\psi/X]$)

for the result of substituting ψ for x (resp. X) in φ, where bound variables are implicitly renamed to prevent variable capture. We define the following logical constructs as syntactic sugar in the usual way:

$$\neg\varphi \equiv \bot \to \varphi \qquad \varphi_1 \vee \varphi_2 \equiv \neg\varphi_1 \to \varphi_2 \qquad \varphi_1 \wedge \varphi_2 \equiv \neg(\neg\varphi_1 \vee \neg\varphi_2)$$

$$\top \equiv \neg\bot \qquad \forall x.\, \varphi \equiv \neg\exists x.\, \neg\varphi \qquad \nu X.\, \varphi \equiv \neg\mu X.\, \neg\varphi[\neg X/X]$$

We assume the standard precedence between these logical constructs.

2.2 Matching Logic Semantics

ML has a *pattern matching* semantics. Patterns are interpreted on a given underlying carrier set, and each pattern is interpreted as a *set* of the elements that *match* the pattern.

Definition 2.3. Given $\mathbb{\Sigma} = (EV, SV, \Sigma)$, a $\mathbb{\Sigma}$-*model* (or simply *model*) is a tuple $(M, \_\bullet\_, \{M_\sigma\}_{\sigma\in\Sigma})$, where

1. M: a carrier set, required to be nonempty;
2. $\_\bullet\_ \colon M \times M \to \mathcal{P}(M)$ is a function, called the *interpretation of application*; here, $\mathcal{P}(M)$ is the powerset of M;
3. $M_\sigma \subseteq M$ is a subset of M, the *interpretation of σ in M*, for each $\sigma \in \Sigma$.

By abuse of notation, we write M to denote the above model.

For notational simplicity, we extend $\_\bullet\_$ from over elements to over sets, *pointwisely*, as follows:

$$\_\bullet\_ \colon \mathcal{P}(M) \times \mathcal{P}(M) \to \mathcal{P}(M) \qquad\qquad A \bullet B = \bigcup_{a\in A, b\in B} a \bullet b \ \text{ for } A, B \subseteq M$$

Note that $\emptyset \bullet A = A \bullet \emptyset = \emptyset$ for any $A \subseteq M$.

Next, we define variable valuations and pattern interpretations:

Definition 2.4. Given $\mathbb{\Sigma} = (EV, SV, \Sigma)$ and a model M, an M-*valuation* (or simply *valuation*) is a function $\rho \colon (EV \cup SV) \to (M \cup \mathcal{P}(M))$ that maps element variables to elements in M and set variables to subsets of M; that is, $\rho(x) \in M$ for all $x \in EV$ and $\rho(X) \subseteq M$ for all $X \in SV$. We define *pattern interpretation* $|\_|_\rho \colon \textsc{Pattern} \to \mathcal{P}(M)$ inductively as follows:

$$|x|_\rho = \{\rho(x)\} \quad |X|_\rho = \rho(X) \quad |\sigma|_\rho = M_\sigma \quad |\bot|_\rho = \emptyset \quad |\varphi_1\, \varphi_2|_\rho = |\varphi_1|_\rho \bullet |\varphi_2|_\rho$$

$$|\varphi_1 \to \varphi_2|_\rho = M \setminus (|\varphi_1|_\rho \setminus |\varphi_2|_\rho) \quad |\exists x.\, \varphi|_\rho = \bigcup_{a\in M} |\varphi|_{\rho[a/x]} \quad |\mu X.\, \varphi|_\rho = \mu\mathcal{F}^\rho_{X,\varphi}$$

where $\rho[a/x]$ is the valuation ρ' such that $\rho'(x) = a$, $\rho'(y) = \rho(y)$ for any $y \in EV$ distinct from x, and $\rho'(X) = \rho(X)$ for any $X \in SV$. Here, $\mathcal{F}^\rho_{X,\varphi} \colon \mathcal{P}(M) \to \mathcal{P}(M)$ is the function defined as $\mathcal{F}^\rho_{X,\varphi}(A) = |\varphi|_{\rho[A/X]}$ for every $A \subseteq M$, where $\rho[A/X]$ is the valuation ρ' such that $\rho'(X) = A$, $\rho'(Y) = \rho(Y)$ for any $Y \in SV$ distinct from X, and $\rho'(x) = \rho(x)$ for any $x \in EV$. By structural induction we can prove that $\mathcal{F}^\rho_{X,\varphi}$ is a monotone function (see Exercise 2.5). Therefore, $\mathcal{F}^\rho_{X,\varphi}$ has a unique least fixpoint which we denote as $\mu\mathcal{F}^\rho_{X,\varphi}$, by the Knaster-Tarski fixpoint theorem [7] (see Exercise 2.6).

Exercise 2.5. Prove that $\mathcal{F}^\rho_{X,\varphi}$ as defined in Definition 2.4 is a monotone function for all X, φ, ρ; that is, $\mathcal{F}^\rho_{X,\varphi}(A) \subseteq \mathcal{F}^\rho_{X,\varphi}(B)$ whenever $A \subseteq B$.

Exercise 2.6. Prove that $\mathcal{F}^\rho_{X,\varphi}$ has a unique least fixpoint given as below:

$$\mu\mathcal{F}^\rho_{X,\varphi} = \bigcap \left\{ A \subseteq M \mid \mathcal{F}^\rho_{X,\varphi}(A) \subseteq A \right\}$$

Hint: Use the Knaster-Tarski fixpoint theorem [7].

The following proposition shows that the interpretation of φ only depends on the valuations of the free variables of φ.

Proposition 2.7. *For any pattern φ and two valuations ρ_1, ρ_2, if $\rho_1(x) = \rho_2(x)$ for all $x \in FV(\varphi)$, then $|\varphi|_{\rho_1} = |\varphi|_{\rho_2}$.*

Explanation. Hint: By structural induction on φ. □

In particular, given a model M and a pattern φ, if $FV(\varphi) = \emptyset$, then the interpretation of φ is the same under all valuations. In this case, we use $|\varphi|$ (without the subscript ρ) to denote the (unique) interpretation of φ in the given model M. We call φ a *closed pattern* if $FV(\varphi) = \emptyset$.

Given a model M, matching logic (ML) patterns are interpreted as subsets of M. This is clearly different from the classical first-order logic (FOL), where formulas are interpreted as either true or false. In ML, we can use two special sets, the total set M and the empty set $\emptyset$, to represent the logical true and false. Given M and a pattern φ, we call φ an *M-predicate* iff $|\varphi|_\rho \in \{\emptyset, M\}$ for all ρ. We call φ a *predicate* iff it is an *M-predicate* for all M. Intuitively, a predicate pattern makes "statement". If the statement is a fact, then the predicate pattern is interpreted as M. Otherwise, it is interpreted as $\emptyset$. We will see many examples of predicate patterns in Section 3.

Definition 2.8. For M and φ, we say that M *validates* φ or φ *holds* in M, written $M \vDash \varphi$, iff $|\varphi|_\rho = M$ for all ρ. Let $\Gamma \subseteq \textsc{Pattern}$ be a pattern set. We say that M *validates* Γ, written $M \vDash \Gamma$, iff $M \vDash \psi$ for all $\psi \in \Gamma$. We say that Γ *validates* φ, written $\Gamma \vDash \varphi$, iff $M \vDash \Gamma$ implies $M \vDash \varphi$, for all M.

Definition 2.9. A matching logic *specification* $\mathsf{SPEC} = (EV, SV, \Sigma, \Gamma)$ is a tuple, where (EV, SV, Σ) is a signature and Γ is a set of patterns called *axioms*. We write $\mathsf{SPEC} \vDash \varphi$ to mean $\Gamma \vDash \varphi$ for a pattern φ. We write $M \vDash \mathsf{SPEC}$ to mean $M \vDash \Gamma$ for a model M.

For simplicity, we often do not explicitly mention EV and SV when we define a specification SPEC.

3 Specification Examples: Important Mathematical Instruments

We define several important mathematical instruments such as equality, membership, sorts, functions, predicates, and constructors, as matching logic specifications.

3.1 Definedness Symbol and Related Instruments

Recall that a pattern φ is interpreted as the set of elements that match it. When φ can be matched by at least one element, we say that φ is *defined*. In this section, we will construct from a given φ, a new pattern $\lceil \varphi \rceil$ called the *definedness pattern*, which is a predicate pattern stating that φ is defined.

Definition 3.1. Let $\lceil \_ \rceil$ be a (constant) symbol, which we call the *definedness symbol*. We write $\lceil \varphi \rceil$ as syntactic sugar of $\lceil \_ \rceil\, \varphi$, obtained by applying $\lceil \_ \rceil$ to φ, for any φ. We define the following axiom

$$(\textsc{Definedness}) \quad \lceil x \rceil$$

It is more compact and readable if we write the above definition as a matching logic specification as follows:

spec DEFINEDNESS
  Metavariable: pattern φ, element variable x
  Symbol: $\lceil \_ \rceil$
  Notation:
    $\lceil \varphi \rceil \equiv \lceil \_ \rceil\, \varphi$
  Axiom:
    $(\textsc{Definedness})$ $\lceil x \rceil$
endspec

Here, keyword "Metavariable" introduces the metavariables used in the specification. Keyword "Symbol" enumerates the symbols declared in the specification. Keyword "Notation" introduces notations (syntactic sugar). Keyword "Axiom" lists all axioms (schemas). For readability, some axioms are named. For example, here we name the axiom $\lceil x \rceil$ by (DEFINEDNESS). For simplicity, we feel free to omit Metavariable when they are understood. Therefore, DEFINEDNESS can be presented in the following more compact form:

spec DEFINEDNESS
  Symbol: $\lceil \_ \rceil$
  Notation:
    $\lceil \varphi \rceil \equiv \lceil \_ \rceil \, \varphi$
  Axiom:
    (DEFINEDNESS) $\lceil x \rceil$
endspec

The following proposition explains why $\lceil \_ \rceil$ is called *definedness* symbol.

Proposition 3.2. *Let M be a model such that $M \vDash$ DEFINEDNESS. For any φ and ρ, we have $|\lceil \varphi \rceil|_\rho = M$ iff $|\varphi|_\rho \neq \emptyset$, and $|\lceil \varphi \rceil|_\rho = \emptyset$ iff $|\varphi|_\rho = \emptyset$.*

Exercise 3.3. Prove Proposition 3.2.

Using $\lceil \_ \rceil$, we can define important mathematical instruments as notations. Let us include these notations also in DEFINEDNESS as shown below:

spec DEFINEDNESS
  Symbol: $\lceil \_ \rceil$
  Notation:
    $\lceil \varphi \rceil \equiv \lceil \_ \rceil \, \varphi$ $\lfloor \varphi \rfloor \equiv \neg \lceil \neg \varphi \rceil$ $\varphi_1 = \varphi_2 \equiv \lfloor \varphi_1 \leftrightarrow \varphi_2 \rfloor$
    $x \in \varphi \equiv \lceil x \wedge \varphi \rceil$ $\varphi_1 \subseteq \varphi_2 \equiv \lfloor \varphi_1 \leftrightarrow \varphi_2 \rfloor$ $\varphi_1 \neq \varphi_2 \equiv \neg(\varphi_1 = \varphi_2)$
    $x \notin \varphi \equiv \neg(x \in \varphi)$ $\varphi_1 \not\subseteq \varphi_2 \equiv \neg(\varphi_1 \subseteq \varphi_2)$
  Axiom:
    (DEFINEDNESS) $\lceil x \rceil$
endspec

The following proposition shows that the above mathematical notations have the expected semantics.

Proposition 3.4. *Let M be a model such that $M \vDash$ DEFINEDNESS. For any $x, \varphi, \varphi_1, \varphi_2$ and ρ, we have*

1. $|\lfloor \varphi \rfloor|_\rho = M$ *if* $|\varphi|_\rho = M$*; otherwise,* $|\lfloor \varphi \rfloor|_\rho = \emptyset$*;*
2. $|\varphi_1 = \varphi_2|_\rho = M$ *if* $|\varphi_1|_\rho = |\varphi_2|_\rho$*; otherwise,* $|\varphi_1 = \varphi_2|_\rho = \emptyset$*;*
3. $|x \in \varphi|_\rho = M$ *if* $\rho(x) \in |\varphi|_\rho$*; otherwise,* $|x \in \varphi|_\rho = \emptyset$*;*
4. $|\varphi_1 \subseteq \varphi_2|_\rho = M$ *if* $|\varphi_1|_\rho \subseteq |\varphi_2|_\rho$*; otherwise,* $|\varphi_1 \subseteq \varphi_2|_\rho = \emptyset$*; note that* $|x \subseteq \varphi|_\rho = |x \in \varphi|_\rho$*;*

3.2 Inhabitant Symbol and Related Instruments

ML is an unsorted logic. There is no built-in support in ML for sorts or many-sorted functions. However, we can define sort s as an ML symbol, and use a special symbol $[\![\_]\!]$, called the *inhabitant symbol*, to build the *inhabitant pattern* $[\![\_]\!] \, s$, often written as $[\![s]\!]$, which is a pattern matched by all the elements that have sort s. In this way we can axiomatize sorts and their properties in ML.

Let us first define the following basic specification for sorts:

spec SORTS
  Import: DEFINEDNESS
  Metavariable: pattern φ, element variable $s{:}Sorts$
  Symbol: $[\![\_]\!], Sorts$

Notation:

$[\![s]\!] \equiv [\![\_]\!]\, s$

$\neg_s \varphi \equiv (\neg\varphi) \wedge [\![s]\!]$

$\forall x{:}s.\, \varphi \equiv \forall x.\, (x \in [\![s]\!]) \to \varphi$

$\exists x{:}s.\, \varphi \equiv \exists x.\, (x \in [\![s]\!]) \wedge \varphi$

endspec

Here, keyword "Import" imports all the symbols, notations, and axioms defined in specification DEFINEDNESS. Symbol $[\![\_]\!]$ is called the inhabitant symbol. Symbol *Sorts* is used to represent the sort set. Notation $\neg_s\varphi$ is called *sorted negation*. Intuitively, $\neg_s\varphi$ is matched by all the elements that have sort s and do not match φ. Notations $\forall x{:}s.\, \varphi$ and $\exists x{:}s.\, \varphi$ are called *sorted quantification*, where x only ranges over the elements of sort s.

3.2.1 An Example: Defining Many-Sorted Signatures in Matching Logic

Let us consider a *many-sorted signature* (S, F, Π) and see how to capture it as an ML specification. In (S, F, Π), S is a set of *sorts* denoted $s_1, s_2, \dots,$ $F = \{F_{s_1 \cdots s_n, s}\}_{s_1,\dots,s_n,s \in S}$ is a family set of *many-sorted functions* denoted $f \in F_{s_1 \cdots s_n, s}$, and $\Pi = \{\Pi_{s_1 \cdots s_n}\}_{s_1,\dots,s_n \in S}$ is a family set of *many-sorted predicates* denoted $\pi \in \Pi_{s_1 \cdots s_n}$. For $f \in F_{s_1 \cdots s_n, s}$ and $\pi \in \Pi_{s_1 \cdots s_n}$, we call the sorts $s_1, \dots, s_n$ the *argument sorts*. For $f \in F_{s_1 \cdots s_n, s}$, we call s the *return sort*.

Intuitively, we will define for each $s \in S$ a corresponding ML symbol also denoted s, which represents the *sort name of* s. The inhabitant of s is represented by the inhabitant pattern $[\![s]\!]$. The symbol *Sorts* then includes all sorts $s \in S$. Functions and predicates are represented as symbols, whose arities are axiomatized by ML patterns. This is made formal in the following:

spec MANYSORTED$\{S, F, \Pi\}$

Import: SORTS

Metavariable: $s \in S, f \in F_{s_1 \cdots s_n, s}, \pi \in \Pi_{s_1 \cdots s_n}$

Axiom:

(Sort Name) $(s \in [\![Sorts]\!]) \wedge (\exists z.\, s = z)$

(Nonempty Inhabitant) $[\![s]\!] \neq \bot$

(Function) $\forall x_1{:}s_1 \dots \forall x_n{:}s_n.\, \exists y{:}s.\, f\, x_1 \,\cdots\, x_n = y$

(Predicate) $\forall x_1{:}s_1 \dots \forall x_n{:}s_n.\, \pi\, x_1 \,\cdots\, x_n = \top \vee \pi\, x_1 \,\cdots\, x_n = \bot$

endspec

We explain the above specification. Firstly, MANYSORTED$\{S, F, \Pi\}$ is a parametric specification and can be instantiated by different many-sorted signatures (S, F, Π). We use s, f, π as metavariables that range over S, F, Π, respectively, and define a corresponding ML symbol for each of them.

Axiom (Sort Name) has two effects. Firstly, it specifies that s belongs to the inhabitant of *Sorts*. Secondly, it specifies that s is a *functional pattern*, in the sense that its interpretation M_s in any model M is a singleton. In other words, the pattern s can be matched by exactly one element, as denoted by the element variable z. This is intended, because conceptually s denotes the sort name s, which is a single "element" in the underlying carrier set of M.

Axiom (Nonempty Inhabitant) specifies that the inhabitant of s is nonempty. Axiom (Function) specifies that $f\, x_1 \,\cdots\, x_n$ is matched by exactly one element y of sort s, given that $x_1, \dots, x_n$ have sorts $s_1, \dots, s_n$. In other words, f is a *many-sorted function* from $s_1, \dots, s_n$ to s. Similarly, (Predicate) specifies that π is a *many-sorted predicate* on $s_1, \dots, s_n$, because it always returns $\top$ or $\bot$. For notational simplicity, we use the function notation $f\colon s_1 \times \cdots \times s_n \to s$ to mean (Function). When $n = 0$, we write $f\colon \epsilon \to s$.

3.2.2 More Instruments about Sorts

The flexibility of ML allows us to easily define various instruments and properties about sorts using ML patterns. In this section we show two more examples: (sorted) partial functions and subsorting.

A partial function $f\colon s_1 \times \cdots \times s_n \rightharpoonup s$ can be undefined on one or more of its arguments. In ML partial functions can be axiomatized by the following axiom:

$$(\textsc{Partial Function}) \quad \forall x_1{:}s_1. \ldots \forall x_n{:}s_n. \exists y{:}s. \, f\, x_1 \, \cdots \, x_n \subseteq y$$

which specifies that $f\, x_1 \cdots x_n$ can be matched by *at most one* element. The undefinedness of f on arguments $x_1, \ldots, x_n$ is captured by $f\, x_1 \cdots x_n$ returning $\bot$. For notational simplicity, we use the partial function notation $f\colon s_1 \times \cdots \times s_n \rightharpoonup s$ to mean the axiom (Partial Function), and when $n = 0$ we write $f\colon \epsilon \rightharpoonup s$.

Subsorting is a partial ordering $\leq$ on the sort set S. When $s_1 \leq s_2$, we say s_1 is a subsort of s_2, and require that the inhabitant of s_1 is a subset of the inhabitant of s_2. Subsorting can be axiomatized in ML as follows:

$$(\textsc{Subsorting}) \quad [\![s_1]\!] \subseteq [\![s_2]\!]$$

ML has a pattern matching semantics. Therefore, the pattern $\sigma\, x_1 \cdots x_n$ can be matched by zero, one, or more elements. As we have defined above, σ is called a function iff $\sigma\, x_1 \cdots x_n$ is matched by one element; it is called a partial function iff $\sigma\, x_1 \cdots x_n$ is matched by at most one element. However, we often do not want to specify the number of elements that match $\sigma\, x_1 \cdots x_n$, but only want to require that all elements that match $\sigma\, x_1 \cdots x_n$ must have sort s, whenever $x_1, \ldots, x_n$ have sorts $s_1, \ldots, s_n$. In this case we call σ a *sorted symbol* and axiomatize it by the following axiom:

$$(\textsc{Sorted Symbol}) \quad \sigma\, [\![s_1]\!] \cdots [\![s_n]\!] \subseteq [\![s]\!]$$

Notation 3.5. Let s be a sort and M be a model, the interpretation $|[\![s]\!]|$ is the inhabitant of s in M. For notational simplicity, we write $[\![s]\!]_M$ as an abbreviation of $|[\![s]\!]|$.

3.3 Constructors and the Inductive Domains

Constructors are extensively used in building programs, data, and semantic structures, in order to define and reason about languages and programs. They can be characterized in the "no junk, no confusion" spirit [8].[1] Specifically, let *Term* be a distinguished sort for terms and $C = \{c_1, c_2, \ldots\}$ be a set of *constructors*. For each c_i, we associate an arity $\mathit{arity}(c_i) \in \mathbb{N}$. We define the following ML specification:

spec CONSTRUCTORS$\{C\}$
  Import: MANYSORTED$\{\{\mathit{Term}\}, C, \emptyset\}$
  Metavariable: $c, d \in C$
  Axiom:
    (No Confusion) where $n = \mathit{arity}(c), m = \mathit{arity}(d)$
      $\forall x_1{:}\mathit{Term} \ldots \forall x_n{:}\mathit{Term}. \forall y_1{:}\mathit{Term} \ldots \forall y_m{:}\mathit{Term}.$
        $\neg(c\, x_1 \cdots x_n \wedge d\, y_1 \cdots y_m)$
      $\forall x_1{:}\mathit{Term} \ldots \forall x_n{:}\mathit{Term}. \forall y_1{:}\mathit{Term} \ldots \forall y_n{:}\mathit{Term}.$
        $(c\, x_1 \cdots x_n \wedge c\, y_1 \cdots y_n) \to (c\, (x_1 \wedge y_1) \cdots (x_n \wedge y_n))$
    (Inductive Domain)
      $[\![\mathit{Term}]\!] = \mu X. \bigvee_{c \in C} c\, X \cdots X$ with n_c X's, where $n_c = \mathit{arity}(c)$
endspec

Note that CONSTRUCTORS$\{C\}$ imports symbols and axioms from the many-sorted specification MANYSORTED$\{\{\mathit{Term}\}, C,$ Intuitively, axiom (No Confusion) says that different constructs build different things and that constructors are injective. Axiom (Inductive Domain) says the inhabitant of *Term* is the smallest set that is closed under all constructors.

[1] The material shown in this section answers a question asked by Jacques Carette on the *mathoverflow* site (`https://mathoverflow.net/questions/16180/formalizing-no-junk-no-confusion`) ten years ago: Are there logics in which these requirements ("no junk, no confusion") can be internalized?

Proposition 3.6. *Let M be any model such that $M \vDash \mathsf{CONSTRUCTORS}\{C\}$. Let $[\![Term]\!]_M = |[\![Term]\!]|$ be the inhabitant of Term in M (see Notation 3.5). For any $c \in C$ with arity $n = arity(c)$, we define a function*

$$f_c \colon \underbrace{[\![Term]\!]_M \times \cdots \times [\![Term]\!]_M}_{n\ times} \to \mathcal{P}([\![Term]\!]_M)$$

as follows:

$$f_c(a_1, \dots, a_n) = (\cdots (M_c \bullet a_1) \bullet \cdots \bullet a_n), \quad \text{for } a_1, \dots, a_n \in [\![Term]\!]_M$$

Then we have $f_c(a_1, \dots, a_n)$ is a singleton for every $a_1, \dots, a_n \in [\![Term]\!]_M$.

Explanation. By the axiom (Function) for $c \in C$, defined in the specification $\mathsf{MANYSORTED}\{\{Term\}, C, \emptyset\}$. $\square$

Remark 3.7. Since $f_c(a_1, \dots, a_n)$ is a singleton that contains exactly one element, we abuse the notation and denote *that element* also as $f_c(a_1, \dots, a_n)$. Since f_c is fully determined by M_c and the interpretation of application $\_\bullet\_$ given by M, we abuse the notation and write $M_c(a_1, \dots, a_n)$ to mean $f_c(a_1, \dots, a_n)$, when M is given.

Proposition 3.8. *Let M be any model such that $M \vDash \mathsf{CONSTRUCTORS}\{C\}$. Let distinct $c, d \in C$, $n = arity(c)$, $m = arity(d)$. We define functions (see Remark 3.7):*

$$M_c \colon \underbrace{[\![Term]\!]_M \times \cdots \times [\![Term]\!]_M}_{n\ times} \to [\![Term]\!]_M$$
$$M_d \colon \underbrace{[\![Term]\!]_M \times \cdots \times [\![Term]\!]_M}_{m\ times} \to [\![Term]\!]_M$$

Then we have that M_c, M_d are injective functions, and their ranges are disjoint.

Explanation. By axioms (No Confusion). $\square$

Proposition 3.9. *Let $\mathbb{T}\mathrm{erm}$ be the set of terms built from constructors in C. Then for any model $M \vDash \mathsf{CONSTRUCTORS}\{C\}$, we have that $[\![Term]\!]_M$ is isomorphic to $\mathbb{T}\mathrm{erm}$.*

Explanation. By axiom (Inductive Domain). $\square$

4 Matching Logic Proof System

In this section we review the Hilbert-style proof system for matching logic given in [2]. The proof system is shown in Fig. 1. We write $\mathsf{SPEC} \vdash \varphi$ to mean that φ can be proved by the proof system using the axioms in SPEC. The following theorem shows that the proof system is sound.

Theorem 4.1 ([2])**.** $\mathsf{SPEC} \vdash \varphi$ *implies* $\mathsf{SPEC} \vDash \varphi$.

In this paper we will use the proof system to simplify our reasoning about ML validity and semantics. The following derived rules are useful for coinductive reasoning:

(Post-Fixpoint) $\quad \vdash \nu X.\, \varphi \to \varphi[\nu X.\, \varphi / X]$

(Knaster-Tarski) $\quad \dfrac{\vdash \psi \to \varphi[\psi/X]}{\vdash \psi \to \nu X.\, \varphi}$

FOL Reasoning	(TAUTOLOGY)	φ if φ is a propositional tautology over patterns
	(MODUS PONENS)	$\dfrac{\varphi_1 \quad \varphi_1 \to \varphi_2}{\varphi_2}$
	($\exists$-QUANTIFIER)	$\varphi[y/x] \to \exists x.\, \varphi$
	($\exists$-GENERALIZATION)	$\dfrac{\varphi_1 \to \varphi_2}{(\exists x. \varphi_1) \to \varphi_2}$ if $x \notin FV(\varphi_2)$
Technical Rules	(EXISTENCE)	$\exists x.\, x$
	(SINGLETON)	$\neg\, (C_1[x \wedge \varphi] \wedge C_2[x \wedge \neg\varphi])$
Frame Reasoning	(PROPAGATION$_\perp$)	$C[\perp] \to \perp$
	(PROPAGATION$_\vee$)	$C[\varphi_1 \vee \varphi_2] \to C[\varphi_1] \vee C[\varphi_2]$
	(PROPAGATION$_\exists$)	$C[\exists x.\, \varphi] \to \exists x.\, C[\varphi]$ if $x \notin FV(C)$
	(FRAMING)	$\dfrac{\varphi_1 \to \varphi_2}{C[\varphi_1] \to C[\varphi_2]}$
Fixpoint Reasoning	(SUBSTITUTION)	$\dfrac{\varphi}{\varphi[\psi/X]}$
	(PRE-FIXPOINT)	$\varphi[\mu X.\, \varphi/X] \to \mu X.\, \varphi$
	(KNASTER-TARSKI)	$\dfrac{\varphi[\psi/X] \to \psi}{\mu X.\, \varphi \to \psi}$

Figure 1: A Hilbert-style proof system of matching logic [2] (where $C[\varphi], C_1[\varphi], C_2[\varphi]$ denote patterns of the form $\varphi\,\psi$ or $\psi\,\varphi$ for some ψ)

5 Understanding Models and Interpretation of Patterns

In this section we explain, based on an example, the flexibility to define models for ML specifications and how various patterns are interpreted in a model. Let us consider the following specification of natural numbers (we present the complete specification for clarity):

spec BNAT
  Symbol: $\lceil \_ \rceil, [\![\_]\!], \mathit{Sorts}, \mathit{Nat}, \mathbb{0}, \mathbb{s}, \mathit{le}$
  Axiom:
    (DEFINEDNESS) :
      $\forall x.\, \lceil x \rceil$
    (SORT NAME) :
      $\mathit{Nat} \colon \epsilon \to \mathit{Sorts}$
    (FUNCTION) :
      $\mathbb{0} \colon \epsilon \to \mathit{Nat}$
      $\mathbb{s} : \mathit{Nat} \to \mathit{Nat}$
    (PREDICATE) :
      $\forall x{:}\mathit{Nat}.\, \mathit{le}\, x = \top \vee \mathit{le}\, x = \bot$
endspec

5.1 Three Matching Logic Models of the Specification BNAT

We present three possible models for the specification BNAT. The first model is the canonical model of natural numbers, the second one is related to the greatest fixpoint, and the third one is similar to the first one but with a less conventional interpretation for $\mathbb{s}$ and le.

The First Matching Logic Model $\mathbb{M}1$ of BNAT The first model that we will construct for the specification BNAT is based on the standard model of natural numbers.

model $\mathbb{M}1$ **of** BNAT
  Carrier Set M includes:
    $\mathbb{def}, \mathbb{inh}, \mathbb{Nat}, \mathbb{s}, \mathbb{le}$
    $n,$ for $n \in \mathbb{N}$ where $\mathbb{N}$ is the set of natural numbers
    $\mathbb{le} \rightsquigarrow n,$ for $n \in \mathbb{N},$ denoting partial evaluation results
  Symbol Interpretation:
    $\mathbb{M}1_{\lceil\_\rceil} = \{\mathbb{def}\}$ $\mathbb{M}1_{[\![\_]\!]} = \{\mathbb{inh}\}$ $\mathbb{M}1_{Sorts} = \{\mathbb{Nat}\}$
    $\mathbb{M}1_{Nat} = \{\mathbb{Nat}\}$ $\mathbb{M}1_{\mathbb{0}} = \{0\}$ $\mathbb{M}1_{\mathbb{s}} = \{\mathbb{s}\}$ $\mathbb{M}1_{le} = \{\mathbb{le}\}$
  Application Interpretation:
    $\mathbb{def} \bullet a = M,$ for all $a \in M$
    $\mathbb{inh} \bullet \mathbb{Nat} = \mathbb{N}$
    $\mathbb{s} \bullet n = \{n+1\},$ for all $n \in \mathbb{N}$
    $\mathbb{le} \bullet n = \{\mathbb{le} \rightsquigarrow n\},$ for all $n \in \mathbb{N}$
    $(\mathbb{le} \rightsquigarrow n) \bullet m = M,$ if $n \leq m$ for $n, m \in \mathbb{N}$
    $a \bullet b = \emptyset,$ if none of the above applies, for $a, b \in M$
endmodel

The Second Matching Logic Model $\mathbb{M}2$ of BNAT This differs from $\mathbb{M}1$ in that we interpret the inhabitant of *Nat* as the set of *co-natural numbers* $\mathbb{N} \cup \{\infty\}$.

model $\mathbb{M}2$ **of** BNAT
  Carrier Set M includes:
    $\mathbb{def}, \mathbb{inh}, \mathbb{Nat}, \mathbb{s}, \mathbb{le}$
    $n,$ for $n \in \mathbb{N}$ where $\mathbb{N}$ is the set of natural numbers
    $\infty,$ a distinguished infinity symbol
    $\mathbb{le} \rightsquigarrow n,$ for $n \in \mathbb{N}$
    $\mathbb{le} \rightsquigarrow \infty$
  Symbol Interpretation:
    $\mathbb{M}1_{\lceil\_\rceil} = \{\mathbb{def}\}$ $\mathbb{M}1_{[\![\_]\!]} = \{\mathbb{inh}\}$ $\mathbb{M}1_{Sorts} = \{\mathbb{Nat}\}$
    $\mathbb{M}1_{Nat} = \{\mathbb{Nat}\}$ $\mathbb{M}1_{\mathbb{0}} = \{0\}$ $\mathbb{M}1_{\mathbb{s}} = \{\mathbb{s}\}$ $\mathbb{M}1_{le} = \{\mathbb{le}\}$
  Application Interpretation:
    $\mathbb{def} \bullet a = M,$ for all $a \in M$
    $\mathbb{inh} \bullet \mathbb{Nat} = \mathbb{N} \cup \{\infty\}$
    $\mathbb{s} \bullet n = \{n+1\},$ for all $n \in \mathbb{N}$
    $\mathbb{s} \bullet \infty = \{\infty\}$
    $\mathbb{le} \bullet n = \{\mathbb{le} \rightsquigarrow n\},$ for all $n \in \mathbb{N}$
    $\mathbb{le} \bullet \infty = \{\mathbb{le} \rightsquigarrow \infty\}$
    $(\mathbb{le} \rightsquigarrow n) \bullet m = M,$ if $n \leq m$ for $n, m \in \mathbb{N}$
    $(\mathbb{le} \rightsquigarrow n) \bullet \infty = M,$ if $n \in \mathbb{N}$
    $(\mathbb{le} \rightsquigarrow \infty) \bullet \infty = M$
    $a \bullet b = \emptyset,$ if none of the above applies, for $a, b \in M$
endmodel

The Third Matching Logic Model $\mathbb{M}3$ of BNAT This is a less usual model. The purpose of showing it is to show that we may have exotic models:

model $\mathbb{M}3$ **of** BNAT
Carrier Set M includes:
$\mathbb{def}, \mathbb{inh}, \mathbb{Nat}, \mathbb{s}, \mathbb{le}$
r, for $r \in \mathbb{R}_{\geq 0}$ where $\mathbb{R}_{\geq 0}$ is the set of non-negative real numbers
$\mathbb{le} \rightsquigarrow r$, for $r \in \mathbb{R}_{\geq 0}$
Symbol Interpretation:
$\mathbb{M}1_{\lceil \_ \rceil} = \{\mathbb{def}\}$ $\mathbb{M}1_{[\![\_]\!]} = \{\mathbb{inh}\}$ $\mathbb{M}1_{Sorts} = \{\mathbb{Nat}\}$
$\mathbb{M}1_{Nat} = \{\mathbb{Nat}\}$ $\mathbb{M}1_{\mathbb{0}} = \{0\}$ $\mathbb{M}1_{\mathbb{s}} = \{\mathbb{s}\}$ $\mathbb{M}1_{le} = \{\mathbb{le}\}$
Application Interpretation:
$\mathbb{def} \bullet a = M$, for all $a \in M$
$\mathbb{inh} \bullet \mathbb{Nat} = \mathbb{R}_{\geq 0}$
$\mathbb{s} \bullet r = \{r + 1\}$, for all $r \in \mathbb{R}_{\geq 0}$
$\mathbb{le} \bullet r = \{\mathbb{le} \rightsquigarrow r\}$, for all $r \in \mathbb{R}_{\geq 0}$
$(\mathbb{le} \rightsquigarrow r_1) \bullet r_2 = M$, if $r_1 \leq r_2$ for $r_1, r_2 \in \mathbb{R}_{\geq 0}$
$a \bullet b = \emptyset$, if none of the above applies, for $a, b \in M$
endmodel

5.2 Explaining the Interpretation of Patterns in the Three Models

In order to understand how patterns are interpreted in a model, we consider the following four BNAT-patterns: $\mathbb{s}\,\mathbb{0}$, $\neg_{Nat}(\mathbb{s}\,\mathbb{0})$, $x \wedge le\,(\mathbb{s}\,\mathbb{0})\,x$, $\exists x{:}Nat.\, x \wedge le\,(\mathbb{s}\,\mathbb{0})\,x$, and we interpret them in $\mathbb{M}1, \mathbb{M}2, \mathbb{M}3$, respectively. Recall that $\neg_{Nat}(\mathbb{s}\,\mathbb{0}) \equiv [\![Nat]\!] \wedge \neg(\mathbb{s}\,\mathbb{0})$ is the *sorted negation* of $\mathbb{s}\,\mathbb{0}$ within Nat. We shall write $|\varphi|_{\mathbb{M}1,\rho}$ to denote the interpretation of φ in $\mathbb{M}1$. Similarly, we write $|\varphi|_{\mathbb{M}2,\rho}$ and $|\varphi|_{\mathbb{M}3,\rho}$ to mean the interpretation of φ in $\mathbb{M}2$ and $\mathbb{M}3$, respectively.

Interpreting $\mathbb{s}\,\mathbb{0}$ Since this is a closed pattern with no free variables, its interpretation is fully determined by the model and does not depend on the valuations. Let ρ be any valuation. We have:

$$|\mathbb{s}\,\mathbb{0}|_{\mathbb{M}1,\rho} = |\mathbb{s}\,\mathbb{0}|_{\mathbb{M}2,\rho} = |\mathbb{s}\,\mathbb{0}|_{\mathbb{M}3,\rho} = \{1\}$$

Interpreting $\neg_{Nat}(\mathbb{s}\,\mathbb{0})$ This pattern is the negation of $\mathbb{s}\,\mathbb{0}$ within sort Nat:

$$\begin{aligned}
|\mathbb{s}\,\mathbb{0}|_{\mathbb{M}1,\rho} &= \mathbb{N} \setminus \{1\} \\
|\mathbb{s}\,\mathbb{0}|_{\mathbb{M}2,\rho} &= (\mathbb{N} \cup \{\infty\}) \setminus \{1\} \\
|\mathbb{s}\,\mathbb{0}|_{\mathbb{M}3,\rho} &= \mathbb{R}_{\geq 0} \setminus \{1\}
\end{aligned}$$

Interpreting $x \wedge le\,(\mathbb{s}\,\mathbb{0})\,x$ This pattern has a free variable x, so its interpretation depends on the valuation of x. Let ρ be an valuation. Note that if $\rho(x)$ is not in the inhabitant of Nat, then $le\,(\mathbb{s}\,\mathbb{0})\,x$ is undefined (i.e., returning $\bot$), and thus $x \wedge le\,(\mathbb{s}\,\mathbb{0})\,x$ returns $\bot$. This is shown below:

$$|x \wedge le\,(\mathbb{s}\,\mathbb{0})\,x|_{\mathbb{M},\rho} = \emptyset, \text{for } \mathbb{M} \in \{\mathbb{M}1, \mathbb{M}2, \mathbb{M}3\} \text{ and } \rho(x) \notin [\![Nat]\!]_{\mathbb{M}}$$

Recall that $[\![Nat]\!]_{\mathbb{M}}$ is the inhabitant of Nat in $\mathbb{M}$, defined in Notation 3.5.

Next, we consider the case where $\rho \in [\![Nat]\!]_{\mathbb{M}}$, for $\mathbb{M} \in \{\mathbb{M}1, \mathbb{M}2, \mathbb{M}3\}$. Let us consider two valuations as an example: $\rho_0(x) = 0$ and $\rho_3(x) = 3$. Then:

$$\begin{aligned}
|x \wedge le\,(\mathbb{s}\,\mathbb{0})\,x|_{\mathbb{M}1,\rho_0} &= \{\rho_0(x)\} \cap |le\,(\mathbb{s}\,\mathbb{0})\,x|_{\mathbb{M}1,\rho_0} = \{0\} \cap \emptyset = \emptyset \\
|x \wedge le\,(\mathbb{s}\,\mathbb{0})\,x|_{\mathbb{M}2,\rho_0} &= |x \wedge le\,(\mathbb{s}\,\mathbb{0})\,x|_{\mathbb{M}3,\rho_0} = \emptyset, \text{for the same reason as above}
\end{aligned}$$

$$|x \wedge le\,(\mathtt{s}\,\mathbb{0})\,x|_{\mathbb{M}1,\rho_3} = \{\rho_3(x)\} \cap |le\,(\mathtt{s}\,\mathbb{0})\,x|_{\mathbb{M}1,\rho_3} = \{3\} \cap M = \{3\}$$
$$|x \wedge le\,(\mathtt{s}\,\mathbb{0})\,x|_{\mathbb{M}2,\rho_3} = |x \wedge le\,(\mathtt{s}\,\mathbb{0})\,x|_{\mathbb{M}3,\rho_3} = \{3\}, \text{for the same reason as above}$$

Here, we use M to denote the carrier set of $\mathbb{M}$ for $\mathbb{M} \in \{\mathbb{M}1, \mathbb{M}2, \mathbb{M}3\}$.

As we can see from the above, the intuition of $x \wedge le\,(\mathtt{s}\,\mathbb{0})\,x$ is that it equals x if $\mathtt{s}\,\mathbb{0}$ is less than (or equal to) x, and it equals $\emptyset$, otherwise. With this intuition in mind, we can interpret $\exists x{:}Nat.\, x \wedge le\,(\mathtt{s}\,\mathbb{0})\,x$, as shown below.

Interpreting $\exists x{:}Nat.\, x \wedge le\,(\mathtt{s}\,\mathbb{0})\,x$ This is a closed pattern. However, the quantifier $\exists x.\, Nat$ requires us to consider all valuations of x in $[\![Nat]\!]_{\mathbb{M}}$ for $\mathbb{M} \in \{\mathbb{M}1, \mathbb{M}2, \mathbb{M}3\}$.

Let us first consider the interpretation in $\mathbb{M}1$, where $[\![Nat]\!]_{\mathbb{M}1} = \mathbb{N}$:

$$\begin{aligned}|\exists x{:}Nat.\, x \wedge le\,(\mathtt{s}\,\mathbb{0})\,x|_{\mathbb{M}1,\rho} &= \bigcup_{n\in\mathbb{N}} |x \wedge le\,(\mathtt{s}\,\mathbb{0})\,x|_{\rho[n/x],\mathbb{M}1} \\ &= (\{0\} \cap \emptyset) \cup (\{1\} \cap M) \cup (\{2\} \cap M) \cup \cdots \\ &= \{1, 2, \ldots\} \\ &= \mathbb{N} \setminus \{0\}\end{aligned}$$

Next, let us consider the interpretation in $\mathbb{M}2$, where $[\![Nat]\!]_{\mathbb{M}2} = \mathbb{N} \cup \{\infty\}$:

$$\begin{aligned}&|\exists x{:}Nat.\, x \wedge le\,(\mathtt{s}\,\mathbb{0})\,x|_{\mathbb{M}2,\rho} \\ &= \left(\bigcup_{n\in\mathbb{N}} |x \wedge le\,(\mathtt{s}\,\mathbb{0})\,x|_{\rho[n/x],\mathbb{M}2}\right) \cup |x \wedge le\,(\mathtt{s}\,\mathbb{0})\,x|_{\rho[\infty/x],\mathbb{M}2} \\ &= (\mathbb{N} \setminus \{0\}) \cup \{\infty\} \\ &= \mathbb{N} \cup \{\infty\} \setminus \{0\}\end{aligned}$$

Finally, let us consider the interpretation in $\mathbb{M}3$, where $[\![Nat]\!]_{\mathbb{M}3} = \mathbb{R}_{\geq 0}$:

$$|\exists x{:}Nat.\, x \wedge le\,(\mathtt{s}\,\mathbb{0})\,x|_{\mathbb{M}3,\rho} = \bigcup_{r\in\mathbb{R}_{\geq 0}} |x \wedge le\,(\mathtt{s}\,\mathbb{0})\,x|_{\rho[r/x],\mathbb{M}3} = \{r \in \mathbb{R}_{\geq 0} \mid r \geq 1\}$$

Note that the above is not a countable set.

6 Explaining the General Principles of Induction and Coinduction

In this section we explain how the (Knaster-Tarski) proof rule supplies a (co)induction proof principle in ML. The explanation is based on the well-known (co)induction principle expressed in the lattice theory.

6.1 Induction Principle in Complete Lattices and in Matching Logic

There is a clear similarity between the induction principle and the Knaster-Tarski proof rule:

Complete Lattices	Matching Logic
$\dfrac{\mathcal{F}(X) \subseteq X}{\mu\mathcal{F} \subseteq X}$ (IndPrinc)	$\dfrac{\varphi[\psi/X] \to \psi}{\mu X\,.\,\varphi \to \psi}$ (Knaster-Tarski)

The induction principle (IndPrinc) uses a monotonic function $\mathcal{F}$ over a complete lattice. For instance, the functional $\mathcal{F}$ for the natural numbers is given by $\mathcal{F}(X) = \{\mathbb{0}\} \cup \{\mathtt{s}\,x \mid x \in X\}$, defined over the powerset lattice. The set of natural numbers, defined in this way, is $\mu\mathcal{F} = \{\mathbb{0}, \mathtt{s}\,\mathbb{0}, \mathtt{s}^2\,\mathbb{0}, \ldots\}$. A set X satisfying the hypothesis of (IndPrinc) is usually called *pre-fixpoint*.

Explanation. We start by explaining first how (IndPrinc) is used to prove properties. Assume we have to prove a property of the form $\forall x{:}\mu\mathcal{F}.\,\phi(x)$, i.e., *all elements in an inductive set (i.e., a set that is defined as the least fixpoint of $\mathcal{F}$) have property ϕ.* We consider the set $X_\phi = \{x \mid \phi(x)\}$ and we first show that $\mathcal{F}(X_\phi) \subseteq X_\phi$, then applying (IndPrinc) we obtain $\mu\mathcal{F} \subseteq X_\phi$, which is equivalent to say that $\forall x{:}\mu\mathcal{F}.\,\phi(x)$ holds. For the natural numbers, $\mathcal{F}(X_\phi) \subseteq X_\phi$ is equivalent to $\{\mathbb{0}\} \cup \{\mathtt{s}(x) \mid x \in X_\phi\} \subseteq X_\phi$, i.e., we have to check $\phi(\mathbb{0})$ (base case) and that $\phi(x) \implies \phi(\mathtt{s}(\mathbb{0}))$ (induction step).

Now we explain how (Knaster-Tarski) supplies an induction proof principle in ML. The least fixpoint μF is specified by a pattern $\mu X.\,\varphi$ and the set X_ϕ is specified by the pattern $\psi \equiv \exists x.\, x \wedge \phi(x)$. The inclusion $\mu\mathcal{F} \subseteq X_\phi$ is specified by the pattern $\mu X\,.\,\varphi \to \psi$ and the inclusion $\mathcal{F}(X_\phi) \subseteq X_\Phi$ by $\varphi[\psi/X] \to \psi$. For the example of the natural numbers, we have that $\varphi \equiv \mathbb{0} \vee \mathtt{s}\,X$ and $\varphi[\psi/X] \equiv \mathbb{0} \vee \mathtt{s}\,\psi$. It follows that $\varphi[\psi/X] \to \psi$ is equivalent to $\mathbb{0} \to \psi$ and $\mathtt{s}\,\psi \to \psi$, which can be informally expressed as $\psi(0)$ and $\psi(x) \to \psi(\mathtt{s}\,x)$. $\square$

Examples of inductive proofs for natural numbers using the proof rule (Knaster-Tarski) are included in Sections 7.1.2 and 7.2.3. Examples about inductive reasoning for parametric lists are included in Section 7.2.4.

6.2 Coinduction Principle in Complete Lattices and in Matching Logic

The coinduction principle is dual to the induction principle:

Complete Lattices	MmL
$\dfrac{X \subseteq \mathcal{F}(X)}{X \subseteq \nu\mathcal{F}}$ (CoindPrinc)	$\dfrac{\psi \to \varphi[\psi/X]}{\psi \to \nu X\,.\,\varphi}$ (Knaster-Tarski)

A set X satisfying the hypothesis of (CoindPrinc) is usually called *post-fixpoint.* We consider the example of the infinite lists $\nu\mathcal{F} = \{b_0 :: b_1 :: b_2 :: \ldots \mid b_i = \mathbb{0} \vee b_i = \mathbb{1}\}$, where $\mathcal{F}(X) = \{b :: x \mid x \in X, b = \mathbb{0} \vee b = \mathbb{1}\}$, and $b :: x$ is a sugar syntax for $cons\, b\, x$.

Explanation. (CoindPrinc) is used to prove thar $X_\phi \subseteq \nu\mathcal{F}$, i.e., *the set of elements satisfying ϕ is a subset of the coinductive set νF.* For instance, if $\phi(x)$ is $x = b :: x_1 \wedge x_1 = (\mathbb{1} - b) :: x_2 \wedge \phi(x_2) \wedge b \in \mathbb{0} \vee \mathbb{1}$, then $X_\phi \subseteq \nu\mathcal{F}$ says that the elments having the property ϕ are infinite lists. Note that this is not trivial; we can prove it by (CoindPrinc) and showing that $X_\phi \subseteq \mathcal{F}(X_\phi) = \{y \mid y = b :: x \wedge x \in X_\phi\}$.

Let us explain the above reasoning in ML terms. The coinductive set $\nu\mathcal{F}$ is specified by the pattern $\nu Y.\,\varphi$, where $\varphi \equiv (\mathbb{0} :: Y \vee \mathbb{1} :: Y)$, and X_ϕ is expressed by a pattern ψ defined in the same way as for inductive case: $\psi \equiv \exists x.\, x \wedge \phi(x)$. The inclusion $X_\phi \subseteq \nu\mathcal{F}$ is expressed by $\psi \to \nu Y.\,\varphi$, and the inclusion $X_\phi \subseteq \mathcal{F}(X_\phi)$ is expressed by $\psi \to \varphi[\psi/X]$. For the example of infinite lists, this means that $\psi \to \mathbb{0} :: \psi \vee \mathbb{1} :: \psi$.

The usual coinduction proof rule is explained in plain English as follows: In order to prove that $X_\phi \subseteq \nu\mathcal{F}$,

1. find a subset X;
2. show that X is a post-fixed point: $X \subseteq \mathcal{F}(X)$;
3. show that $X_\phi \subseteq X$.

The same coinduction proof rule is expressed in ML terms as follows: In inorder to prove that $\mathcal{F} \models \psi \to \nu X.\,\varphi$,

1. find a suitable pattern ψ';
2. show that ψ' is a "post-fixed point": $\mathcal{F} \models \psi' \to \varphi[\psi'/X]$;
3. show that $\mathcal{F} \models \psi \to \psi'$.

$\square$

Examples of coinductive proofs are given in Section 7.2.5 and Section 8.

7 Defining Dependent Types as Matching Logic Specifications

Dependent types (sorts) are types whose definitions depend on a value. In this section, we show how to define dependent types as ML specifications.

7.1 Simple Types

We start with the basic types such as Boolean values and natural numbers.

7.1.1 Booleans

spec BOOL
  Import: SORTS
  Symbol: $Bool, tt, ff, !, \&$
  Metavariable: φ_1, φ_2 patterns
  Notation: $\varphi_1 \,\&\, \varphi_2 \equiv \& \, \varphi_1 \, \varphi_2$
  Axiom:
    (Sort Name): $Bool \in [\![Sorts]\!]$
    (Function):
      $ff : \epsilon \to Bool$ $\qquad tt : \epsilon \to Bool$
      $! : Bool \to Bool$ $\qquad \& : Bool \times Bool \to Bool$
    (Inductive Domain): $[\![Bool]\!] = tt \vee ff$
    (No Confusion): $\neg(tt \wedge ff)$
    (Definition):
      $!\, tt = ff$ $\qquad !\, ff = tt$
      $\forall x{:}Bool.\, x \,\&\, tt = x$ $\qquad \forall x{:}Bool.\, x \,\&\, ff = ff$
      $\forall x{:}Bool.\, tt \,\&\, x = x$ $\qquad \forall x{:}Bool.\, ff \,\&\, x = ff$
endspec

Explanation. The type/sort *Bool* has two constant constructors *tt* and *ff*, which are specified as functional constants. Therefore, in any model $M \vDash$ BOOL, the inhabitant of *Bool* in M must be a set consisting of exactly two elements: the interpretation of *tt* and the interpretation of *ff*. The axioms that define ! and & are usual. □

7.1.2 Natural Numbers

spec NAT
  Import: SORTS
  Symbol: $Nat, \mathbb{0}, \mathbb{s}$
  Axiom:
    (Sort Name): $Nat \in [\![Sorts]\!]$
    (Function):
      $\mathbb{0} : \epsilon \to Nat$ $\qquad \mathbb{s} : Nat \to Nat$
    (Inductive Domain): $[\![Nat]\!] = \mu X.\, \mathbb{0} \vee \mathbb{s}\, X$
    (No Confusion):
      $\forall x{:}Nat.\, \neg(\mathbb{0} \wedge \mathbb{s}\, x)$
      $\forall x, y{:}Nat.\, \mathbb{s}\, x \wedge \mathbb{s}\, y \to \mathbb{s}(x \wedge y)$
endspec

Therefore, *Nat* is the smallest set built from $\mathbb{0}$ and $\mathbb{s}$, which are the only two constructs of *Nat*.

Proposition 7.1. *The following propositions hold:*

1. NAT $\models \mathbb{0} \in [\![Nat]\!]$
2. NAT $\models suc\,[\![Nat]\!] \subseteq [\![Nat]\!]$
3. NAT $\models \forall x{:}Nat.\, \mathbb{0} \neq \mathbb{s}\, x$
4. NAT $\models \forall x{:}Nat.\, y{:}Nat.\, \mathbb{s}\, x \neq y \to x \neq \mathbb{s}\, y$

Explanation. We prove Item 1 as an example and leave the rest as exercises. Let M be any model such that $M \vDash \mathsf{NAT}$. Recall that the axiom (Function) $\mathbb{0}\colon \epsilon \to Nat$ is a shortcut of $\exists z.\, z \in [\![Nat]\!] \wedge z = \mathbb{0}$. Therefore, for some (irrelevant) valuation ρ we have $|\exists z.\, z \in [\![Nat]\!] \wedge z = \mathbb{0}|_\rho = \bigcup_{a \in M} |z \in [\![Nat]\!] \wedge z = \mathbb{0}|_{\rho[a/z]} = M$. Note that $|z \in [\![Nat]\!] \wedge z = \mathbb{0}|_{\rho[a/z]} \in \{0, M\}$ for all a. Therefore, there exists $a_0 \in M$ such that $|z \in [\![Nat]\!] \wedge z = \mathbb{0}|_{\rho[a_0/z]} = M$. Note that $|z \in [\![Nat]\!] \wedge z = \mathbb{0}|_{\rho[a_0/z]} = |z \in [\![Nat]\!]|_{\rho[a_0/z]} \cap |z = \mathbb{0}|_{\rho[a_0/z]} = M$ implies that $a_0 \in [\![Nat]\!]_M$ and $\{a_0\} = M_\mathbb{0}$. Therefore, $|\mathbb{0} \wedge [\![Nat]\!]|_\rho = |\mathbb{0}|_\rho \cap |[\![Nat]\!]|_\rho = M_\mathbb{0} \cap [\![Nat]\!]_M \neq \emptyset$, and thus, $|\mathbb{0} \in [\![Nat]\!]|_\rho = |\lceil \mathbb{0} \wedge [\![Nat]\!] \rceil|_\rho = M$. Since M is any model with $M \vDash \mathsf{NAT}$, we conclude that $\mathsf{NAT} \vDash \mathbb{0} \in [\![Nat]\!]$. □

Exercise 7.2. Prove Items 2-4 in Proposition 7.1.

Proposition 7.3. *For any $M \vDash \mathsf{NAT}$, let $[\![Nat]\!]_M = |[\![Nat]\!]|_M$ be the inhabitant of Nat in M. Then we have that $[\![Nat]\!]_M$ is isomorphic to $\mathbb{N}$, where $\mathbb{N}$ is the set of natural numbers.*

Explanation. Let $M_\mathbb{0}$ and $M_\mathbb{s}$ be the interpretations of $\mathbb{0}$ and $\mathbb{s}$ in M, respectively. By the axiom (Function) for $\mathbb{0}$, we know that $M_\mathbb{0}$ is a singleton, whose element we denote (by abuse of notation) as $\mathbb{0}$. By the axiom (Function) for $\mathbb{s}$, we know that for for any $n \in [\![Nat]\!]_M$, $M_\mathbb{s} \bullet n$ is a singleton, whose element we denote (by abuse of notation) as $\mathbb{s}(\mathbb{0})$. By the axiom (No Confusion), we have that the elements $\mathbb{0}, \mathbb{s}(\mathbb{0}), \mathbb{s}(\mathbb{s}(\mathbb{0})), \dots$ are all distinct. Clearly, the set $\{\mathbb{0}, \mathbb{s}(\mathbb{0}), \mathbb{s}(\mathbb{s}(\mathbb{0})), \dots\}$ is isomorphic to $\mathbb{N}$, and by abuse of notation we use $\mathbb{N}$ to denote the set. Next, we prove that $[\![Nat]\!]_M$ is isomorphic to $\mathbb{N}$. By the axiom (Inductive Domain), $[\![Nat]\!]_M = |\mu X.\, \mathbb{0} \vee \mathbb{s}\, X|_\rho = \mu\mathcal{F}$, where $\mathcal{F}\colon \mathcal{P}(M) \to \mathcal{P}(M)$ is defined as $\mathcal{F}(A) = |\mathbb{0} \vee \mathbb{s}\, X|_{\rho[A/X]} = \{\mathbb{0}\} \cup \{\mathbb{s}(n) \mid n \in A\}$. Then $\mathcal{F}(\mathbb{N}) = \{\mathbb{0}\} \cup \{\mathbb{s}(n) \mid n \in \mathbb{N}\} = \mathbb{N}$, so $\mathbb{N}$ is a fixpoint of $\mathcal{F}$. On the other hand, we can prove by induction that any fixpoint of $\mathcal{F}$ includes $\mathbb{s}(\cdots(\mathbb{s}(\mathbb{0}))\cdots)$ with any number of $\mathbb{s}$. Therefore, $\mathbb{N}$ is indeed the least fixpoint of $\mathcal{F}$, and thus $[\![Nat]\!]_M$ is isomorphic to $\mathbb{N}$. □

Proposition 7.4 (Successor Prefixpoint)**.** *Let P be a set variable. Then we have*

1. $\mathsf{NAT} \models (\mathbb{s}\, P \to P) \leftrightarrow (\forall x.\, x \in P \to \mathbb{s}\, x \in P)$;
2. $\mathsf{NAT} \models P \subseteq [\![Nat]\!] \to ((\mathbb{s}\, P \to P) \leftrightarrow (\forall x{:}Nat.\, x \in P \to \mathbb{s}\, x \in P))$.

We call both equivalences (PrefixSucc). *Note that in Item 1 we use the unsorted quantification $\forall x$ while in Item 2 we use the sorted quantification $\forall x{:}Nat$.*

Explanation. We only explain Item 1 as an example. Let us assume a model $M \vDash \mathsf{NAT}$ and a valuation ρ. Note that $\forall x.\, x \in P \to \mathbb{s}\, x \in P$ is a predicate pattern. Then we have that

$$\begin{aligned} & |\mathbb{s}\, P \to P|_\rho = M \\ \text{iff}\quad & |\mathbb{s}\, P|_\rho \subseteq |P|_\rho \\ \text{iff}\quad & M_\mathbb{s} \bullet |P|_\rho \subseteq |P|_\rho \\ \text{iff}\quad & M_\mathbb{s} \bullet n \in |P|_\rho \text{ for all } n \in |P|_\rho \\ \text{iff}\quad & |\forall x.\, x \in P \to \mathbb{s}\, x \in P|_\rho = M \end{aligned}$$

The similar reasoning holds for $|\mathbb{s}\, P \to P|_\rho = \emptyset$. □

Proposition 7.5 (Peano Induction)**.** *Let P be a set variable.*

$$\mathsf{NAT} \models P \subseteq [\![Nat]\!] \to ((\mathbb{0} \in P \wedge (\mathbb{s}\, P \to P)) \to \forall x{:}Nat.\, x \in P) \qquad (\text{IndNat})$$

Explanation. Let $M \vDash \mathsf{NAT}$ and ρ by any valuation. If $\rho(P) \not\subseteq [\![Nat]\!]_M$, then $|P \subseteq [\![Nat]\!]|_\rho = \emptyset$, and thus $|P \subseteq [\![Nat]\!] \to ((\mathbb{0} \in P \wedge (\mathbb{s}\, P \to P)) \to \forall x{:}Nat.\, x \in P)|_\rho = M$. Therefore, we assume $\rho(P) \subseteq [\![Nat]\!]_M$, and our goal is to prove that $|(\mathbb{0} \in P \wedge (\mathbb{s}\, P \to P)) \to \forall x{:}Nat.\, x \in P|_\rho = M$.

If $|\mathbb{0} \in P|_\rho = \emptyset$ or $|s\, P \to P|_\rho = \emptyset$, we have $|(\mathbb{0} \in P \wedge (\mathbb{s}\, P \to P)) \to \forall x{:}Nat.\, x \in P|_\rho = M$. Therefore, we assume that $|\mathbb{0} \in P|_\rho = |s\, P \to P|_\rho = M$; that is, $\mathbb{0} \in \rho(P)$, and by Proposition 7.1, for all $[\![Nat]\!]_M \in \mathbb{N}$, $\mathbb{s}(n) \in \rho(P)$. By Proposition 7.3, we have $\rho(P) = [\![Nat]\!]_M$, and thus $|\forall x{:}Nat.\, x \in P|_\rho = M$. □

Remark 7.6. Let $\varphi(x)$ be a FOL formula with a distinguished variable x. Let set variable P be matched by exactly the elements x such that $\varphi(x)$ holds. Then clearly, we have that $\varphi(x)$ holds if and only if $x \in P$. Based on this observation, we can rewrite (IndNat) in the following more familiar form:

$$\Gamma \models \varphi(0) \wedge (\forall y{:}Nat.\, \varphi(y) \to \varphi(\mathbb{s}\, y)) \to \forall x{:}Nat.\, \varphi(x)$$

7.2 Parameterized Types

A parameterized type (sort) is a type that depends on other type values. In this section we define five parameterized types: product types, sum (co-product) types, function types, parametric (finite) lists, and parametric streams (infinite-lists). The key observation is that since ML is an unsorted logic and sorts are definable concepts, it is natural and straightforward to define parameterized types by defining proper sorts axioms.

7.2.1 Product Types

Given two sorts s_1 and s_2 we define a new sort $s_1 \otimes s_2$, called the *product (sort) of* s_1 *and* s_2, as follows:

spec PROD$\{s_1, s_2\}$
  Import: SORTS
  Symbol: $\otimes, \langle\_,\_\rangle, \pi_1, \pi_2$
  Notation:
    $s_1 \otimes s_2 \equiv \otimes\, s_1\, s_2$
    $\langle\_,\_\rangle\, x\, y \equiv \langle x, y\rangle$
  Axiom:
    (Product Sort)
      $s_1 \in [\![Sorts]\!] \wedge s_2 \in [\![Sorts]\!] \to s_1 \otimes s_2 \in [\![Sorts]\!]$
    (Pair)
      $\langle\_,\_\rangle : s_1 \times s_2 \to s_1 \otimes s_2$
    (Project Left)
      $\pi_1 : s_1 \otimes s_2 \to s_1$
    (Project Right)
      $\pi_2 : s_1 \otimes s_2 \to s_2$
    (Injection)
      $\langle x_1, x_2\rangle = \langle y_1, y_2\rangle \to x_1 = x_2 \wedge y_1 = y_2$
    (Inverse PairProj1)
      $\forall x_1{:}s_1.\, \forall x_2{:}s_2.\, \pi_i\, \langle x_1, x_2\rangle = x_i,\ i = 1, 2$
    (Inverse PairProj2)
      $\forall y{:}s_1 \otimes s_2.\, \langle \pi_1\, y, \pi_2\, y\rangle = y$
endspec

Explanation. Axioms (Pair), (Project Left), and (Project Right) are instances of the axiom schema (Function). Axioms (Inverse PairProj1) and (Inverse PairProj2) express the fact that the pair function and the projections are inverse with respect to each other. □

Fact 7.7. The following hold:

1. $\forall y{:}s_1 \otimes s_2.\, \exists\, x_1{:}s_1.\, \exists\, x_2{:}s_2.\, y = \langle x_1, x_2\rangle$.
2. $[\![s_1 \otimes s_2]\!] = [\![s_1]\!] \times [\![s_2]\!]$.

Explanation. (Item 1). Consider $x_i = \pi_i\, y,\ i = 1, 2$. We obtain $x_i \in [\![s_i]\!],\ i = 1, 2$, by the coresponding (Project _) axiom. The equality $y = \langle x_1, x_2\rangle$ follows by (Inverse PairProj2).
(Item 2). The pair function $\langle\_,\_\rangle$ is a bijection by (Injection) and Item 1. □

7.2.2 Sum (Coproduct) Types

Given two sorts s_1 and s_2 we define a new sort $s_1 \oplus s_2$, called the *sum (coproduct) of* s_1 *and* s_2, as follows:

spec SUM$\{s_1, s_2\}$
  Import: SORTS
  Symbol: $\oplus, \iota_1, \iota_2, \epsilon_1, \epsilon_2$

Notation: $s_1 \oplus s_2 \equiv \oplus\, s_1\, s_2$
Axiom:
(INJECT LEFT)
$s_1 \oplus s_2 \in [\![Sorts]\!] \iota_1 : s_1 \to s_1 \oplus s_2$
(INJECT RIGHT)
$\iota_2 : s_2 \to s_1 \oplus s_2$
(EJECT LEFT)
$\epsilon_1 : s_1 \oplus s_2 \rightharpoonup s_1$
(EJECT RIGHT)
$\epsilon_2 : s_1 \oplus s_2 \rightharpoonup s_2$
(INVERSE INJEJ1)
$\forall x{:}s_i.\, \epsilon_i\, (\iota_i\, x) = x,\ i = 1, 2$
(INVERSE INJEJ2)
$\forall x{:}s_{3-i}.\, \epsilon_i\, (\iota_{3-i}\, x) = \bot,\ i = 1, 2$
(COPRODUCT)
$\forall s_1, s_2{:}Sorts.\, [\![s_1 \oplus s_2]\!] \subseteq (\iota_1\, [\![s_1]\!]) \vee (\iota_2\, [\![s_2]\!])$
(DISJ)
$\forall s_1, s_2{:}Sorts.\, (\iota_1\, [\![s_1]\!]) \wedge (\iota_2\, [\![s_2]\!]) = \bot$
endspec

Explanation. (INJECT _) and (EJECT _) are instances of (FUNCTION) and (PARTIAL FUNCTION), respectively. □

Fact 7.8. The following hold:

1. ι_1 and ι_2 are injective functions.
2. $[\![s_1 \oplus s_2]\!] = (\iota_1\, [\![s_1]\!]) \vee (\iota_2\, [\![s_2]\!])$.

Explanation. 1. Take ι_1 as an example. Suppose $\iota_1\, x = \iota_1\, y$, then we have $\epsilon_1(\iota_1\, x) = \epsilon_1(\iota_1\, y)$; by (INVERSEINJEJ1), we have $x = y$.
2. We have to show that $[\![s_1 \oplus s_2]\!] \supseteq (\iota_1\, [\![s_1]\!]) \vee (\iota_2\, [\![s_2]\!])$, which follows by (INJECT _). □

Fact 7.9. $[\![s_1 \oplus s_2]\!] = [\![s_1]\!] \uplus [\![s_2]\!]$, where $\uplus$ denotes set disjoint union, defined as $[\![s_1]\!] \uplus [\![s_2]\!] = ([\![s_1]\!] \times \{1\}) \cup ([\![s_2]\!] \times \{2\})$.

Explanation. Formally, we need to establish the following bijection:

$$\iota \colon [\![s_1 \oplus s_2]\!] \to [\![s_1]\!] \uplus [\![s_2]\!]$$
$$\epsilon \colon [\![s_1]\!] \uplus [\![s_2]\!] \to [\![s_1 \oplus s_2]\!]$$

Note that by (COPRODUCT), for every $b \in [\![s_1 \oplus s_2]\!]$, there exists $i \in \{1, 2\}$, such that $b \in \iota_i([\![s_i]\!])$; by the injectivity of ι_i, we know there exists a unique $a_b \in [\![s_i]\!]$ such that $b = \iota_i(a_b)$. Then, we define ι as follows:

$$\iota(b) = \begin{cases} (a_b, 1) & \text{if } a_b \in [\![s_1]\!] \text{ such that } b = \iota_i(a_b) \\ (a_b, 2) & \text{if } a_b \in [\![s_2]\!] \text{ such that } b = \iota_i(a_b) \end{cases}$$

Then, we define ϵ as follows:

$$\epsilon((a, i)) = \iota_i(a)$$

It is straightforward to see that ι and ϵ are inverse to each other. This proves that $[\![s_1 \oplus s_2]\!] = [\![s_1]\!] \uplus [\![s_2]\!]$. □

7.2.3 Function Types

Given two sorts s_1 and s_2 we define a new sort $s_1 \mathbin{\bigcirc\mkern-15mu\rightarrow} s_2$, called the *function sort from* s_1 *to* s_2, as follows:

spec FUN$\{s_1, s_2\}$
  Import: SORTS
  Symbol: $\mathbin{\bigcirc\mkern-15mu\rightarrow}$
  Notation:
    $s_1 \mathbin{\bigcirc\mkern-15mu\rightarrow} s_2 \equiv \mathbin{\bigcirc\mkern-15mu\rightarrow}\, s_1\, s_2$
    $(f =^{s_1}_{ext} g) \equiv (\forall x{:}s_1.\, f\,x = g\,x)$
  Axiom:
    $s_1 \mathbin{\bigcirc\mkern-15mu\rightarrow} s_2 \in [\![Sorts]\!]$
    $[\![s_1 \mathbin{\bigcirc\mkern-15mu\rightarrow} s_2]\!] = \exists f.\, f \wedge \forall x{:}s_1.\, \exists\, y{:}s_2.\, f\,x = y$
endspec

Fact 7.10. The following hold:

1. $\forall f.\, (\forall x{:}s_1.\, \exists y{:}s_2.\, f\,x = y) \to f \in [\![s_1 \mathbin{\bigcirc\mkern-15mu\rightarrow} s_2]\!]$.
2. $\forall f{:}s_1 \mathbin{\bigcirc\mkern-15mu\rightarrow} s_2.\, (\forall x{:}s_1.\, \exists y{:}s_2.\, f\,x = y)$.

Remark 7.11. Even if strong related, there is a difference between $f{:}s_1 \mathbin{\bigcirc\mkern-15mu\rightarrow} s_2$ and $f : s_1 \to s_2$. The former says that $f \in [\![s_1 \mathbin{\bigcirc\mkern-15mu\rightarrow} s_2]\!]$ and the later is a sugar syntax for the axiom

$$\forall x{:}s_1.\, \exists\, y{:}s_2.\, f\,x = y$$

that is equivalent to

$$\forall x.\, x \in [\![s_1]\!] \to \exists\, y.\, y \in [\![s_2]\!] \wedge f\,x = y \qquad .$$

The relationship between the two notations is easy to see if we note that the definition of $[\![s_1 \mathbin{\bigcirc\mkern-15mu\rightarrow} s_2]\!]$ can be written as $\exists f.\, f \wedge f{:}\, s_1 \to s_2$. However, $f \in [\![s_1 \mathbin{\bigcirc\mkern-15mu\rightarrow} s_2]\!]$ says further that f is a functional pattern.

Since we have axiomatic definitions for the product and respectively function sorts, we may use them to formalize the iteration and recursion principles for the type of natural numbers.

Proposition 7.12 (Natural Numbers Iteration Principle)**.**

$$\begin{array}{c}\forall h.\, \forall c{:}s.\, \forall f{:}s \mathbin{\bigcirc\mkern-15mu\rightarrow} s.\, (h\,\mathbb{0} = c \wedge \forall n{:}Nat.\, h\,(\mathbb{s}\,n) = f\,(h\,n)) \to \\ (\forall n{:}Nat.\, \exists\, y{:}s.\, h\,n = y)\end{array} \tag{\textsc{ItNat}}$$

Explanation. (ItNat) is equivalent to

$$\begin{array}{c}\forall h.\, \forall c{:}s.\, \forall f{:}s \mathbin{\bigcirc\mkern-15mu\rightarrow} s.\, (h\,\mathbb{0} = c \wedge \forall n{:}Nat.\, h\,(\mathbb{s}\,n) = f\,(h\,n)) \to \\ ([\![Nat]\!] \subseteq \exists x.\, \exists\, y{:}s.\, x \wedge h\,x = y)\end{array}$$

and we apply then the induction principle:

$$\dfrac{\dfrac{c{:}s \qquad h\,\mathbb{0} = c}{\mathsf{NAT} \models \mathbb{0} \in \exists x.\, \exists\, y{:}s.\, x \wedge h\,x = y}\;\textsc{Hyp} \qquad \dfrac{\dfrac{\dfrac{f{:}s \mathbin{\bigcirc\mkern-15mu\rightarrow} s}{\mathsf{NAT} \models \begin{array}{l}\exists x.\, \exists\, y{:}s.\, x \wedge h\,x = y \\ \to (\exists x.\, \exists\, y{:}s.\, \mathbb{s}\,x \wedge f\,(h\,x) = y)\end{array}}\;\textsc{Hyp}}{\mathsf{NAT} \models \begin{array}{l}\exists x.\, \exists\, y{:}s.\, x \wedge h\,x = y \\ \to (\exists x.\, \exists\, y{:}s.\, \mathbb{s}\,x \wedge h\,(\mathbb{s}\,x) = y)\end{array}}\;\textsc{Hyp}}{\mathsf{NAT} \models \begin{array}{l}\exists x.\, \exists\, y{:}s.\, x \wedge h\,x = y \\ \to \mathbb{s}\,(\exists x.\, \exists\, y{:}s.\, x \wedge h\,x = y)\end{array}}\;\textsc{Def}\ \mathbb{s}}{\mathsf{NAT} \models [\![Nat]\!] \subseteq \exists x.\, \exists\, y{:}s.\, x \wedge h\,x = y}\;\textsc{IndNat}$$

□

Example 7.13. The following ML specification defines two functions *plus* and *mult* on natural numbers in the usual way:

spec PLUS&MULT
  Import: NAT
  Symbol: $\mathit{plus}, \mathit{mult}$
  Metavariable: element variables $x{:}\mathit{Nat}, y{:}\mathit{Nat}$
  Axiom:
    $\mathit{plus}\, x\, \mathbb{0} = x$
    $\mathit{plus}\, x\, (\mathbb{s}\, y) = \mathbb{s}\, (\mathit{plus}\, x\, y)$
    $\mathit{mult}\, x\, \mathbb{0} = \mathbb{0}$
    $\mathit{mult}\, x\, (\mathbb{s}\, y) = \mathit{plus}\, (\mathit{mult}\, x\, y)\, x$
endspec

The fact that *plus* and *mult* are well-defined follows by applying (ItNat). For instance, for *plus* we consider $h = \mathit{plus}\, x$, $c = \mathbb{0}$, and $= \mathbb{s}$.

Proposition 7.14 (Natural Numbers (Primitive) Recursion Principle)**.**

$$\forall h.\, \forall c{:}s.\, \forall g{:}(s \otimes \mathit{Nat}) \ominus s.\, (h\, \mathbb{0} = c \wedge \forall n{:}\mathit{Nat}.\, h\, (\mathbb{s}\, n) = g\, (h\, n)\, n) \to (\forall n{:}\mathit{Nat}.\, \exists\, y{:}s.\, h\, n = y) \tag{PrRecNat}$$

Explanation. (PrRecNat) is equivalent to

$$\forall h.\, \forall c{:}s.\, \forall g{:}(s \otimes \mathit{Nat}) \ominus s.\, (h\, \mathbb{0} = c \wedge \forall n{:}\mathit{Nat}.\, h\, (\mathbb{s}\, n) = g\, ((h\, n)\, n)) \to (\llbracket \mathit{Nat} \rrbracket \subseteq \exists x.\, \exists\, y{:}s.\, x \wedge h\, x = y)$$

and we apply then the induction principle:

$$\dfrac{\dfrac{c{:}s \qquad h\, \mathbb{0} = c}{\mathsf{NAT} \models \mathbb{0} \in \exists x.\, \exists\, y{:}s.\, x \wedge h\, x = y}\,\textsc{Hyp} \qquad \dfrac{\dfrac{\dfrac{g{:}s \otimes \mathit{Nat} \ominus s}{\mathsf{NAT} \models \begin{array}{l} \exists x.\, \exists\, y{:}s.\, x \wedge h\, x = y \\ \to \\ (\exists x.\, \exists\, y{:}s.\, \mathbb{s}\, x \wedge g\, (h\, x)\, x = y) \end{array}}\,\textsc{Hyp}}{\mathsf{NAT} \models \begin{array}{l} \exists x.\, \exists\, y{:}s.\, x \wedge h\, x = y \\ \to \\ (\exists x.\, \exists\, y{:}s.\, \mathbb{s}\, x \wedge h\, (\mathbb{s}\, x) = y) \end{array}}\,\textsc{Hyp}}{\mathsf{NAT} \models \begin{array}{l} \exists x.\, \exists\, y{:}s.\, x \wedge h\, x = y \\ \to \\ \mathbb{s}\, (\exists x.\, \exists\, y{:}s.\, x \wedge h\, x = y) \end{array}}\,\textsc{Def}\ \mathbb{s}}{\mathsf{NAT} \models \llbracket \mathit{Nat} \rrbracket \subseteq \exists x.\, \exists\, y{:}s.\, x \wedge h\, x = y}\,\textsc{IndNat}$$

□

Example 7.15. The following ML specification defines the factorial function *fact* in the usual way:

spec FACT
  Import: NAT
  Symbol: fact
  Metavariable: element variables $x{:}\mathit{Nat}, y{:}\mathit{Nat}$
  Axiom:
    $\mathit{fact}\, \mathbb{0} = \mathbb{s}\, \mathbb{0}$
    $\mathit{fact}\, (\mathbb{s}\, x) = \mathit{mult}\, (\mathit{fact}\, x)\, x$
endspec

The fact that *fact* is well-defined follows by applying (PrRecNat) with $h = \mathit{fact}$, $c = \mathbb{s}\, \mathbb{0}$, and $g = \mathit{mult}$.

7.2.4 Parameterized (Finite) Lists

Parametric lists is a canonical example of polymorphic datatype, i.e., a datatype parametrized by another type. Polymorphic datatypes are included in almost programming languages (Java, C++, Haskell, etc.), known also as generic types. For instance, in C++ were introduced in 1987, but without rigorously taking into account a logical foundation for their semantics [9]; now generic programming in C++ is redesigned using the semantic notion of *concept*, which is a predicate on template arguments [10]. In this section we present a complete specification for the parametric lists, which can be used as a foundation for any implementation.

Datatype Specification of Lists The most usual way to define the parametric lists is using a BNF-like notation:

$$List\langle Elt\rangle ::= nil \mid cons(Elt, List\langle Elt\rangle)$$

A reader familiar with a functional programming perhaps prefer a Haskell-like notation:

```
data List a = Nil | Cons a (List a)
```

This specification is sufficient for someone who wants to use the datatype, but, for sure, is not sufficient for implementing the datatype.

Matching Logic Specification of Lists The following ML specification of the parametric lists shows how much semantical information is missing from the above specification.

spec LIST$\{s\}$
Import: SORTS
Symbol: $List$
Metavariable: $x{:}s, x'{:}s, \ell{:}List\langle s\rangle, \ell'{:}List\langle s\rangle$ element variables
Notation: $List\langle s\rangle \equiv List\, s$
Axiom:
(Sort Name): $s \in [\![Sorts]\!] \to List\langle s\rangle \in [\![Sorts]\!]$
(Function):
$\exists y.\, List = y$
$\exists y{:}List\langle s\rangle.\, nil = y$
$\exists y{:}List\langle s\rangle.\, cons\, x\, \ell = y$
(Inductive Domain):
$[\![List\langle s\rangle]\!] = \mu X.\, nil \vee cons\, [\![s]\!]\, X$
(No Confusion):
$nil \neq cons\, x\, \ell$
$cons\, x\, \ell = cons\, x'\, \ell' \to cons\, (x \wedge x')\, (\ell \wedge \ell')$
endspec

Explanation. From (Sort Name) we infer that $List\langle s\rangle$ is a functional constant, i.e., $\exists y.\, List\langle s\rangle = y$. Some programming languages may have constraints on polymorphic datatypes. For instance, in Java s cannot be a primitive type. Then the axiom (Sort Name) is replaced by $\exists y.\, List\langle s\rangle \subseteq y$, $List\langle s\rangle \subseteq [\![Sorts]\!]$, and $s \in PrimitiveSorts \to List\langle s\rangle = \bot$. The first axiom (Function) says that the generic name $List$ is a function constant; the next two constraint $cons$ and nil to functional intyerpretation. The axiom (Inductive Domain) says that the set of the inhabitants of $List\langle s\rangle$ contains exactly those elements that we obtain by repeatedly using of finitely times the constructors nil and $cons$. □

The next results show how many interesting properties can be formally derived from the above specification and internally expressed in ML.

Proposition 7.16 (List Induction Principle)**.**

$$\mathsf{LIST}\{s\} \models (nil \in P \wedge cons\, [\![s]\!]\, P \subseteq P) \to [\![List\, s]\!] \subseteq P \qquad \text{(IndList)}$$

Explanation.

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathsf{LIST} \models \mathit{nil} \in P}{\mathsf{LIST} \models \mathit{nil} \to P} \qquad \dfrac{\mathsf{LIST} \models \mathit{cons}\,\llbracket s \rrbracket\, P \subseteq P}{\mathsf{LIST} \models \mathit{cons}(\llbracket s \rrbracket, P) \to P}}{\mathsf{LIST} \models \mathit{nil} \vee \mathit{cons}(\llbracket s \rrbracket, P) \to P}\ \textsc{PropTaut}}{\mathsf{LIST} \models (\mathit{nil} \vee \mathit{cons}(\llbracket s \rrbracket, L))[P/L] \to P}\ \textsc{Replace}}{\mathsf{LIST} \models \mu L{:}\mathit{List}\,.\, \mathit{nil} \vee \mathit{cons}(\llbracket s \rrbracket, L) \to P}\ \textsc{K-T}}{\mathsf{LIST} \models \llbracket \mathit{List}\ s \rrbracket \subseteq P}\ \textsc{Def.}{\subseteq}
$$

□

Remark 7.17. Using a notation similar to that from 7.6, (IndList) can be rewritten in the next more familiar form:

$$
\mathsf{LIST}\{s\} \models \varphi(\mathit{nil}) \wedge (\forall \ell{:}\mathit{List}\ s.\, \varphi(\ell) \to \forall x{:}s.\, \varphi(\mathit{cons}\, x\, \ell)) \to \forall \ell{:}\mathit{List}\ s.\, \varphi(\ell)
$$

Proposition 7.18 (List Iteration Principle)**.**

$$
\begin{aligned}
\mathsf{LIST}\{s\} \models & \forall h.\, \forall c{:}s'.\, \forall f{:}s{\otimes}s' \ominus s'. \\
& (h\, \mathit{nil} = c \wedge \forall x{:}s.\, \forall \ell{:}\mathit{List}\langle s \rangle.\, h\,(\mathit{cons}\, x\, y) = f\,(x, h\,\ell)) \to \\
& (\forall \ell{:}\mathit{List}\langle s \rangle.\, \exists\, y{:}s'.\, h\,\ell = y)
\end{aligned} \qquad (\textsc{ItList})
$$

Explanation. (ItList) is equivalent to

$$
\begin{aligned}
\mathsf{LIST}\{s\} \models & \forall h.\, \forall c{:}s'.\, \forall f{:}s{\otimes}s' \ominus s'. \\
& (h\, \mathit{nil} = c \wedge \forall x{:}s.\, \forall \ell{:}\mathit{List}\langle s \rangle.\, h\,(\mathit{cons}\, x\, y) = f\,(x, h\,\ell)) \to \\
& (\llbracket \mathit{List}\langle s \rangle \rrbracket \subseteq \exists x.\, \exists\, y{:}s'.\, x \wedge h\, x = y)
\end{aligned}
$$

which allows to use the induction principle for list to derive a justification:

$$
\dfrac{\dfrac{c{:}s' \qquad h\,\mathit{nil} = c}{\mathsf{LIST} \models \mathit{nil} \in \begin{array}{c}\exists x.\, \exists\, y{:}s.\, x \\ \wedge \\ h\, x = y\end{array}}\ \textsc{Hyp} \qquad \dfrac{\dfrac{\dfrac{f{:}s{\otimes}s' \ominus s'}{\mathsf{LIST} \models \begin{array}{l}\exists x.\, \exists\, y{:}s'.\, x \wedge h\, x = y \\ \to \\ \exists x.\, \exists\, y{:}s'.\, \exists a{:}s.\ \begin{array}{l}\mathit{cons}\, a\, x\, \wedge \\ f\, a\,(h\, x) = y\end{array}\end{array}}\ \textsc{Hyp}}{\mathsf{LIST} \models \begin{array}{l}\exists x.\, \exists\, y{:}s'.\, x \wedge h\, x = y \\ \to \\ \exists x.\, \exists\, y{:}s'.\, \exists a{:}s.\ \begin{array}{l}\mathit{cons}\, a\, x\, \wedge \\ h\,(\mathit{cons}\, a\, x) = y\end{array}\end{array}}\ \textsc{Hyp}}{\mathsf{LIST} \models \begin{array}{l}\exists x.\, \exists\, y{:}s'.\, x \wedge h\, x = y \\ \to \\ \mathit{cons}\,\llbracket s \rrbracket\,(\exists x.\, \exists\, y{:}s'.\, x \wedge h\, x = y)\end{array}}\ \textsc{Def}\ \mathit{cons}}{\mathsf{LIST} \models \llbracket \mathit{List}\langle s \rangle \rrbracket \subseteq \exists x.\, \exists\, y{:}s'.\, x \wedge h\, x = y}\ \textsc{IndList}
$$

□

A direct use of (ItList) is given by the definition of *map*:

Example 7.19. Let MAP be the following ML specification:

spec MAP
  Import: $\mathsf{LIST}\{s\}$
  Symbol: map
  Metavariable: element variables $x{:}s, \ell{:}\mathit{List}\langle s \rangle, g{:}s \ominus s'$
  Axiom:
    $\mathit{map}\ g\ \mathit{nil} = \mathit{nil}$
    $\mathit{map}\ g\,(\mathit{cons}\ x\ \ell) = \mathit{cons}\,(g\, x)\,(\mathit{map}\ g\ \ell)$
endspec

Then we obtain

$$
\begin{aligned}
&\mathsf{MAP} \models \mathit{map} \in [\![((s \mathrel{\bigcirc\mkern-14mu\rightarrow} s') \otimes \mathit{List}\langle s\rangle) \mathrel{\bigcirc\mkern-14mu\rightarrow} \mathit{List}\langle s'\rangle]\!]\\
&\mathsf{MAP} \models \mathit{map}\, g \in [\![\mathit{List}\langle s\rangle \mathrel{\bigcirc\mkern-14mu\rightarrow} \mathit{List}\langle s'\rangle]\!]
\end{aligned}
$$

by applying the List Iteration Principle with $c = \mathit{nil}$, $h = \mathit{map}\, g$, $f\, x\, \ell' = \mathit{cons}\,(g\, x)\, \ell'$, where x is of sort s and ℓ' of sort $\mathit{List}\langle s'\rangle$.

Proposition 7.20 (Lists (Primitive) Recursion Principle)**.**

$$
\begin{aligned}
\mathsf{LIST} \models\, & \forall h.\, \forall c{:}s'.\, \forall g{:}(s \otimes \mathit{List}\langle s\rangle) \mathrel{\bigcirc\mkern-14mu\rightarrow} s'.\\
& (h\, \mathit{nil} = c \wedge \forall x{:}s.\, \forall \ell{:}\mathit{List}\langle s\rangle.\, h\,(\mathit{cons}\, x\, \ell) = g\,(h\, \ell)\, x\, \ell) \to\\
& (\forall \ell{:}\mathit{List}\langle s\rangle.\, \exists\, y{:}s'.\, h\, \ell' = y)
\end{aligned}
\tag{PrRecList}
$$

Explanation. We write (PrRecList) in the equivalent form

$$
\begin{aligned}
\mathsf{LIST} \models\, & \forall h.\, \forall c{:}s'.\, \forall g{:}(s \otimes \mathit{List}\langle s\rangle) \mathrel{\bigcirc\mkern-14mu\rightarrow} s'.\\
& (h\, \mathit{nil} = c \wedge \forall x{:}s.\, \forall \ell{:}\mathit{List}\langle s\rangle.\, h\,(\mathit{cons}\, x\, \ell) = g\,(h\, \ell)\, x\,)\ell \to\\
& ([\![\mathit{List}\langle s\rangle]\!] \subseteq \exists x.\, \exists\, y{:}s'.\, x \wedge h\, x = y)
\end{aligned}
$$

and apply the inuction principle for lists:

$$
\dfrac{
\dfrac{c{:}s \qquad h\,\mathbb{0} = c}{\mathsf{LIST} \models \mathit{nil} \in \exists x.\, \exists\, y{:}s'.\, x \wedge h\, x = y}\ \textsc{Hyp}
\qquad
\dfrac{
\dfrac{
\dfrac{g{:}(s' \otimes s \otimes \mathit{List}\langle s\rangle) \mathrel{\bigcirc\mkern-14mu\rightarrow} s'}{\mathsf{LIST} \models \begin{array}{l}\exists x.\, \exists\, y{:}s'.\, x \wedge h\, x = y\\ \to\\ \exists x.\, \exists\, y{:}s'.\, \exists a{:}s.\ \begin{array}{l}\mathit{cons}\, a\, x\, \wedge\\ g\,(h\, x)\, a\, x = y\end{array}\end{array}}\ \textsc{Hyp}
}{\mathsf{LIST} \models \begin{array}{l}\exists x.\, \exists\, y{:}s'.\, x \wedge h\, x = y\\ \to\\ \exists x.\, \exists\, y{:}s'.\, \exists a{:}s.\ \begin{array}{l}\mathit{cons}\, a\, x\, \wedge\\ h\,(\mathit{cons}\, a\, x) = y\end{array}\end{array}}\ \textsc{Hyp}
}{\mathsf{LIST} \models \begin{array}{l}\exists x.\, \exists\, y{:}s'.\, x \wedge h\, x = y\\ \to\\ \mathit{cons}\, [\![s]\!]\, (\exists x.\, \exists\, y{:}s'.\, x \wedge h\, x = y)\end{array}}\ \textsc{Def}\mathit{cons}
}{\mathsf{LIST} \models [\![\mathit{List}\langle s\rangle]\!] \subseteq \exists x.\, \exists\, y{:}s'.\, x \wedge h\, x = y}\ \textsc{IndList}
$$

□

Here is a direct use of the primitive recursive principle for lists:

Example 7.21. Let FOLDR be he following ML specification:

> **spec** FOLDR
> Import: LIST$\{s\}$
> Symbol: *foldr*
> Metavariable: element variables $x{:}s, z{:}s', \ell{:}\mathit{List}\langle s\rangle, f{:}s \otimes s' \mathrel{\bigcirc\mkern-14mu\rightarrow} s'$
> Axiom:
> $\mathit{foldr}\ f\ z\ \mathit{nil} = z$
> $\mathit{foldr}\ f\ z\,(\mathit{cons}\, x\, \ell) = f\, x\,(\mathit{foldr}\ f\ z\, \ell)$
> **endspec**

Then we obtain

$$
\begin{aligned}
&\mathsf{FOLDR} \models \mathit{foldr}{:}((s \mathrel{\bigcirc\mkern-14mu\rightarrow} s') \otimes s' \otimes \mathit{List}\langle s\rangle) \mathrel{\bigcirc\mkern-14mu\rightarrow} s'\\
&\mathsf{FOLDR} \models \mathit{foldr}\ f\ z{:}\mathit{List}\langle s\rangle \mathrel{\bigcirc\mkern-14mu\rightarrow} s'
\end{aligned}
$$

by applying the List Primitive Recursion Principle with $c = z'$, $h = \mathit{foldr}\ f\ z$, $g\, y\, x\, \ell = f\, x\, y$, where x is of sort s, y of sort s', and ℓ of sort $\mathit{List}\langle s\rangle$.

7.2.5 Parameterized (Infinite) Streams

Streams (infinite lists) is a canonical example of coinductive type and coinductive reasoning. Infinite datatypes are used in programming languages, e.g., Haskell, together with lazy evaluation, which allows to bypass the undefined values (e.g., the result of an infinite execution of a program).

Infinite Datatype (Codatatype) Specification of Streams Streams can be specified using a BNF-like notation

$$\mathit{Stream}\langle \mathit{Elt}\rangle ::= \mathit{cons}(\mathit{Elt}, \mathit{Stream}\langle \mathit{Elt}\rangle)$$

or a Haskell-like notation:

```
data InfList a
  a ::: (InfList a)
```

where the constructor $x ::: \ell$ corresponds to $\mathit{cons}(x, \ell)$. The constructors of infinite datatypes are useful to define the set of its inhabitants, but useless in practice when we do not need or want runtime pattern-matches on a data constructor which will never occur. Therefore, the equivalent definition with *destructors* is used in practice. For the case of streams, the destructors are hd and tl defined by $hd(\mathit{cons}(x, \ell)) = x$ and $tl(\mathit{cons}(x, \ell)) = \ell$.

Matching Logic Specification of Streams The following ML specification includes both the constructors and the destructors. The constructors are used to define the set of inhabitants as the greatest fixpoint and the destructors are defined axiomatically.

spec STREAM$\{s\}$
  Element Variables:
  Symbol: $\mathit{Stream}, \mathit{cons}, hd, tl, \approx_{\mathit{Stream}}$
  Metavariable: element variables $x, x'{:}s; \ell, \ell_1, \ell_2{:}\mathit{Stream}\langle s\rangle$
  Notation:
    $\mathit{Stream}\langle s\rangle \equiv \mathit{Stream}\, s$
    $\ell_1 \approx_{\mathit{Stream}} \ell_2 \equiv \langle \ell_1, \ell_2\rangle \in \approx_{\mathit{Stream}}$
    $\mathit{cons}\, x\, \langle \ell_1, \ell_2\rangle \equiv \langle \mathit{cons}\, x\, \ell_1, \mathit{cons}\, x\, \ell_2\rangle$
    $\alpha(X) \equiv \exists \ell.\, \ell \wedge hd\, \ell \in [\![s]\!] \wedge tl\, \ell \in X$
    $\beta(R) \equiv \exists \ell, \ell'{:}\mathit{Stream}\langle s\rangle.\, \langle \ell, \ell'\rangle \wedge hd\, \ell = hd\, \ell' \wedge \langle tl\, \ell, tl\, \ell'\rangle \in R$
  Axiom:
    (Sort Name) $\forall s{:}\mathit{Sorts}.\, \mathit{Stream}\langle s\rangle \in [\![\mathit{Sorts}]\!]$
    (Function)
      $\exists y.\, \mathit{Stream} = y$
      $\forall x.\, \forall y.\, \exists z.\, \mathit{cons}\, x\, y = z$
    (Coinductive Domain) $\forall s{:}\mathit{Sorts}.\, [\![\mathit{Stream}\langle s\rangle]\!] = \nu X.\, \mathit{cons}\, [\![s]\!]\, X$
    (No Confusion)
      $\mathit{cons}\, x\, \ell = \mathit{cons}\, x'\, \ell' \to \mathit{cons}\, (x \wedge x')\, (\ell \wedge \ell')$
    (destructors)
      $hd(\mathit{cons}\, x\, \ell) = x$
      $tl(\mathit{cons}\, x\, \ell) = \ell$
    (Bisimilarity)
      $\approx_{\mathit{Stream}} = \nu R{:}\mathit{Stream}\langle s\rangle \otimes \mathit{Stream}\langle s\rangle.\, \mathit{cons}\, [\![s]\!]\, R$
      $\forall \ell_1, \ell_2{:}\mathit{Stream}\langle s\rangle\langle s\rangle.\, (\ell_1 \approx_{\mathit{Stream}} \ell_2) = (\ell_1 = \ell_2)$
endspec

Explanation. The axioms for sorts and constructors are similar to those from finite lists. The notations $\alpha(X)$ and $\beta(R)$ are used to show that we can obtain an equivalent specification using destructors (see below). In order to understand the coinductive definition of the domain, we recall that, given a model M, $|\nu X.\, cons\, [\![s]\!]\, X|_\rho = \nu\mathcal{F}^\rho_{X,\varphi}$, where $\varphi \equiv cons\, [\![s]\!]\, X$. Since $\mathcal{F}^\rho_{X,\varphi}$ is cocontinuous, we have

$$\begin{aligned}
\nu\mathcal{F}^\rho_{X,\varphi} &= M \cap \mathcal{F}^\rho_{X,\varphi}(M) \cap \mathcal{F}^\rho_{X,\varphi}(\mathcal{F}^\rho_{X,\varphi}(M)) \cap \cdots \\
&= M \cap |cons\, [\![s]\!]\, X|_{\rho[M/X]} \cap |cons\, [\![s]\!]\, X|_{\rho[|cons\, [\![s]\!]\, X|_{\rho[M/X]}/X]} \cap \cdots \\
&= M \cap [\![s]\!]_M ::: M \cap [\![s]\!]_M ::: [\![s]\!]_M ::: M \cap \cdots \\
&= [\![s]\!]_M ::: [\![s]\!]_M ::: [\![s]\!]_M ::: \cdots \\
&= \{a_0 ::: a_1 ::: a_2 ::: \cdots \mid a_i \in [\![s]\!]_M, i = 0, 1, 2, \ldots\}
\end{aligned}$$

where $A ::: B \equiv (|cons|_\rho \bullet A) \bullet B$ and $[\![s]\!]_M \equiv |[\![s]\!]|_\rho$. We also have $a_0 ::: a_1 ::: a_2 ::: \cdots \neq b_0 ::: b_1 ::: b_2 ::: \cdots$ if there is i such thar $a_i \neq b_i$, by applying (No Confusion) $i+1$ times.. It is easy to see now the similarity with the Haskel definition of the infinite trees. Note that the definition does not depends on ρ since X is the only variable in φ. Let $[\![s]\!]^\infty_M$ denote this greatest fixpoint.

Another novelty is the inclusion of the bisimilarity in the specification. It is defined similarly to the set of inhabitants, but over pairs of elements. Since we have the additional constraint $R{:}Stream\langle s\rangle{\otimes}Stream\langle s\rangle$, the greatest fixpoint can be computed starting from $[\![s]\!]^\infty_M \times [\![s]\!]^\infty_M$:

$$\begin{aligned}
\nu\mathcal{F}^\rho_{R,\varphi} &= [\![s]\!]^\infty_M \times [\![s]\!]^\infty_M \cap \mathcal{F}^\rho_{R,\varphi}([\![s]\!]^\infty_M \times [\![s]\!]^\infty_M) \cap \mathcal{F}^\rho_{R,\varphi}(\mathcal{F}^\rho_{R,\varphi}([\![s]\!]^\infty_M \times [\![s]\!]^\infty_M)) \cap \cdots \\
&= [\![s]\!]^\infty_M \times [\![s]\!]^\infty_M \cap |cons\, [\![s]\!]\, X|_{\rho[[\![s]\!]^\infty_M \times [\![s]\!]^\infty_M/R]} \cap \cdots \\
&= [\![s]\!]^\infty_M \times [\![s]\!]^\infty_M \cap [\![s]\!]_M ::: [\![s]\!]^\infty_M \times [\![s]\!]^\infty_M \cap [\![s]\!]_M ::: [\![s]\!]_M ::: [\![s]\!]^\infty_M \times [\![s]\!]^\infty_M \cap \cdots \\
&= [\![s]\!]^\infty_M \times [\![s]\!]^\infty_M \cap \{\langle a_0 :: \ell, a_0 :: \ell'\rangle \mid a_0 \in [\![s]\!]_M, \langle \ell, \ell'\rangle \in [\![s]\!]^\infty_M \times [\![s]\!]^\infty_M\} \cap \cdots \\
&= \{\langle a_0 ::: a_1 ::: a_2 ::: \cdots,\ a_0 ::: a_1 ::: a_2 ::: \cdots\rangle \mid a_i \in [\![s]\!]_M, i = 0, 1, 2, \ldots\}
\end{aligned}$$

where φ is now $cons\, [\![s]\!]\, R$. Norte that $[\![s]\!]_M ::: R = \{a ::: R \mid a \in [\![s]\!]_M\} = \{\langle a ::: \ell,\ a ::: \ell'\rangle \mid a \in [\![s]\!]_M, \langle \ell, \ell'\rangle \in R\}$, according to the notation from the specification. ☐

Fact 7.22. The following results show that the streams can be equivalently specified using destructors.

1. $\mathsf{STREAM}\{s\} \models \forall \ell{:}Stream.\ cons(hd\, \ell)\,(tl\, \ell) = \ell$.
2. $\mathsf{STREAM}\{s\} \models [\![Stream]\!] = \nu x.\, \alpha(X)$.
3. $\mathsf{STREAM}\{s\} \models \forall \ell, \ell'{:}Stream.\ cons\,((hd\, \ell) \wedge (hd\, \ell'))\,((tl\, \ell) \wedge (tl\, \ell')) \to \ell \wedge \ell'$.
4. $\mathsf{STREAM}\{s\} \models \approx_{Stream} = \nu R{:}Stream\langle s\rangle{\otimes}Stream\langle s\rangle.\, \beta(R)$

Explanation. Item 1 shows that destructors and constructors are inverse for each other. Item 2 shows that the inhabitant of streams is the biggest set closed under the destructors. Item 3 shows that the constructor *cons* is injective. The name of constructor for *cons* is a bit misused here, because it cannot construct alone streams (there is no a *nil*-like constructor). But it can reconstruct a stream from its components given by the destructors. The notation $\beta(R)$ say that R is a bisimulation (see, e.g., [11]) or a behavioral equivalence (see, e.g., [12]). Then Item 4 specifies that $\approx_{Stream}$ is the largest bisimulation (behavioral equivalence). ☐

Proposition 7.23 (Stream Coinduction Principle I)**.**

$$\mathsf{STREAM}\{s\} \models (P \subseteq P' \wedge P' \subseteq cons\, [\![s]\!]\, P') \to (P \subseteq [\![Stream\langle s\rangle]\!]) \qquad \text{(CoindStream)}$$
$$\mathsf{STREAM}\{s\} \models (R \subseteq R' \wedge R' \subseteq cons\, [\![s]\!]\, R') \to (R \subseteq \approx_{Stream}) \qquad \text{(CoindStreamEqC)}$$

where $P{:}Stream$ and $R : Stream\langle s\rangle{\otimes}Stream\langle s\rangle$.

Explanation. Both are special instances of the coinduction proof rule as discussed at the end of Section 6. For example, there is the proof for $\mathsf{STREAM}\{s\} \models P' \to [\![Stream\langle s\rangle]\!]$, which is equivalent to $\mathsf{STREAM}\{s\} \models P' \subseteq [\![Stream\langle s\rangle]\!]$:

$$\dfrac{\dfrac{P' \to cons\, [\![s]\!]\, P}{P' \to \nu X.\, cons\, [\![s]\!]\, X}\ \text{Knaster-Tarski}}{P' \to [\![Stream\langle s\rangle]\!]}\ \text{Coind Dom}$$

Then we obtain $\mathsf{STREAM}\{s\} \models P \to [\![Stream\langle s\rangle]\!]$ by FOL reasoning. ☐

Corollary 7.24 (Stream Coinduction Principle II)**.**

$$\mathsf{STREAM}\{s\} \models (R \subseteq R' \wedge R' \subseteq \beta(R')) \to (R \subseteq \approx_{\mathit{Stream}}) \qquad (\textsc{CoindStreamEqD})$$

where $R : \mathit{Stream}\langle s\rangle \otimes \mathit{Stream}\langle s\rangle$*.*

Explanation. This coinductive principle is an instance of the coinduction proof rule discussed at the end of Section 6, by observing Item 4 in Fact 7.22. □

Proposition 7.25 (Stream Coiteration Principle)**.**

$$\mathsf{STREAM}\{s\} \models \exists h.\, \exists x{:}s'.\, \exists c{:}s' \mathbin{\bigcirc\!\!\!\!\!\!\to} s.\, \exists g{:}s' \mathbin{\bigcirc\!\!\!\!\!\!\to} s'.\, h(x) \wedge \forall y{:}s'.\, \begin{array}{l} \mathit{hd}\,(h\,y) = c\,y \wedge \\ \mathit{tl}(h\,y) = h\,(g\,y) \end{array} \subseteq [\![\mathit{Stream}\langle s\rangle]\!] \qquad (\textsc{CoitStream})$$

Explanation. Note the use of the existential quantifier, comparing with the dual universal quantifier used in Proposition 7.18. This is due to the fact that c, g, and h are used now to "produce" a set of streams. Let P' denote the pattern

$$h(x) \wedge \forall y{:}s'.\, \mathit{hd}\,(h\,y) = c\,y \wedge \mathit{tl}(h\,y) = h\,(g\,y)$$

and let P denote

$$\exists h.\, \exists x{:}s'.\, \exists c{:}s' \mathbin{\bigcirc\!\!\!\!\!\!\to} s.\, \exists g{:}s' \mathbin{\bigcirc\!\!\!\!\!\!\to} s'.\, P'.$$

We have:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{c \in [\![s' \mathbin{\bigcirc\!\!\!\!\!\!\to} s]\!]}{c\,y\text{ß}[\![s]\!]}\;\textsc{Fol} \qquad \dfrac{g \in [\![s' \mathbin{\bigcirc\!\!\!\!\!\!\to} s']\!]}{h\,(g\,y) \in P}\;\textsc{Fol}}{P' \to \alpha(P)}\;\textsc{Fol}}{P \to \alpha(P)}\;\exists\text{-}\textsc{Generalization}}{P \to \nu X.\, \alpha(X)}\;\textsc{Knaster-Tarski}}{P \to [\![\mathit{Stream}\langle s\rangle]\!]}\;\textsc{Fact 7.22 Item 2}$$

□

Example 7.26. Given the following ML specification:

> **spec** CNST&FROM
> Import: NAT + STREAM$\{\mathit{Nat}\}$
> Symbol: $\mathit{cnst}, \mathit{from}$
> Metavariable: element variables $n{:}\mathit{Nat}$
> Axiom:
> $\mathit{hd}\,(\mathit{cnst}\,n) = n \qquad \mathit{hd}\,(\mathit{from}\,n) = n$
> $\mathit{tl}\,(\mathit{cnst}\,n) = \mathit{cnst}\,n \qquad \mathit{tl}\,(\mathit{from}\,n) = \mathit{from}\,(\mathtt{s}\,n)$
> **endspec**

we obtain

$$\mathsf{CNST\&FROM} \vDash \exists\, n{:}\mathit{Nat}.\, \mathit{cnst}\,n \vee \mathit{from}\,n \subseteq [\![\mathit{Stream}\langle s\rangle]\!]$$

by applying CoitStream. For instance, for *from* we take $c\,n = n$ and $g\,n = \mathtt{s}\,n$.

7.3 Fixed-Length Vector Types

A vector type (sort) $\mathit{Vec}\,s\,n$ is a dependent type taking two parameters, where s is the base sort and n denotes the size of the vectors. In this section we will define two versions of vectors. In the first version, vectors of size $n+1$ are built by *pairing* one element and a vector of size n. In the second version, a vector of size $n+1$ is obtained by constructing a *list* whose head is an element and whose tail is a vector of size n. In both versions we require that there is only one vector, the empty vector *null*, whose size is zero.

The First Definition of Vectors Let us first show the first version.

spec VEC1
Import: NAT
Symbol: $\mathit{Vec}, \mathit{null}$
Metavariable: element variables $s{:}\mathit{Sorts},\ n{:}\mathit{Nat}$
Axiom:
(Sort Name): $\forall n{:}\mathit{Nat}.\, \forall s{:}\mathit{Sorts}.\ \mathit{Vec}\, s\, n \in [\![\mathit{Sorts}]\!]$
(Function):
$\exists y.\ \mathit{Vec} = y$
$\exists y.\ \mathit{null} = y$
(Inductive Domain):
$[\![\mathit{Vec}\, s\, \mathbb{0}]\!] = \mathit{null}$
$[\![\mathit{Vec}\, s\, (\mathbb{s}\, n)]\!] = [\![s \otimes \mathit{Vec}\, s\, n]\!]$
endspec

Explanation. (Sort Name) and (Function) are similar to the specifications of lists discussed in Section 7.2.4. The first (Inductive Domain) axiom specifies that there is only one vector *null* whose size is zero. The second (Inductive Domain) axiom specifies that the vector type $\mathit{Vec}\, s\, (\mathbb{s}\, n)$ is an alias for the product type of s and the vector type $\mathit{Vec}\, s\, n$. In other words, a vector type is a nested product type. □

The Second Definition of Vectors Now we define the second version of vectors using finite lists.

spec VEC2
imports : NAT
Symbol: $\mathit{Vec}, \mathit{null}, \mathit{cons}$
Metavariable: element variables $s{:}\mathit{Sorts},\ n{:}\mathit{Nat}$
Axiom:
(Sort Name): $\forall n{:}\mathit{Nat}.\, \forall s{:}\mathit{Sorts}.\ \mathit{Vec}\, s\, n \in [\![\mathit{Sorts}]\!]$
(Function):
$\exists y.\ \mathit{Vec} = y$
$\exists y.\ \mathit{null} = y$
(Inductive Domain):
$[\![\mathit{Vec}\, s\, \mathbb{0}]\!] = \mathit{null}$
$[\![\mathit{Vec}\, s\, (\mathbb{s}\, n)]\!] = \mathit{cons}\, [\![s]\!]\, [\![\mathit{Vec}\, s\, n]\!]$
(No Confusion)
$\forall x{:}s.\, \forall y{:}\mathit{Vec}\ s\, n.\ \mathit{null} \neq \mathit{cons}\, x\, y$
$\forall x, x'{:}s..\, \forall y, y'{:}\mathit{Vec}\, s\, n.\ \mathit{cons}\, x\, y = \mathit{cons}\, x'\, y' \to \mathit{cons}\, (x \wedge x')\, (y \wedge y')$
endspec

Explanation. (Sort Name), (Function), and the first (Inductive Domain) axioms are similar to the first definition version. The second (Inductive Domain) axiom specifies that the vector type $\mathit{Vec}\, s\, (\mathbb{s}\, n)$ contains all the finite lists of length $\mathbb{s}\, n$ where the base sort is s. □

7.4 Dependent Product Types

A dependent product type $\Pi x{:}s_1.\, s_2$ is an extension of the function type $s_1 \ominus\!\!\to s_2$. Let us assume s_2 is an expression where x occurs free. If a function f has the function type $s_1 \ominus\!\!\to s_2$, then for an element a of sort s_1, the term $f\, a$ has sort s_2 no matter what a is. However, if a function f has the dependent product type $\Pi x{:}s_1.\, s_2$, then for an element a of sort s_1, the term $f\, a$ has sort $s_2[a/x]$, which is *dependent* on the argument a. Clearly, if s_2 has no free occurrences of x, then $\Pi x{:}s_1.\, s_2$ reduces to $s_1 \ominus\!\!\to s_2$.

It is straightforward to specify the inhabitant of $\Pi x{:}s_1.\, s_2$ following the similar definition of the inhabitant of $s_1 \ominus s_2$. However, it is (surprisingly) not a trivial task to capture the *binding behavior* of $\Pi x{:}s_1.\, s_2$, in which x in bound in s_2. We shall leave this as an open problem. In this paper we only show how to specify the inhabitant of $\Pi x{:}s_1.\, s_2$.

spec DPROD
  Notation:
    $[\![\Pi x{:}s_1.\, s_2(x)]\!] \equiv \exists f.\, f \wedge \forall x{:}s_1.\, \exists y{:}s_2(x).\, f\, x = y$
endspec

Explanation. In the above notation definition we write $s_2(x)$ to emphasize that x may occur free in s_2. The reader can verify that when $x \notin FV(s_2)$, the above notation reduces to the definition of the inhabitant of the function type $s_1 \ominus s_2$. □

Fact 7.27. The following hold:

1. $\forall f.\, (\forall x{:}s_1.\, \exists y{:}s_2(x).\, f\, x = y) \to f \in [\![\Pi x{:}s_1.\, s_2(x)]\!]$.
2. $\forall f{:}(\Pi x{:}s_1.\, s_2(x)).\, x \in [\![s_1]\!] \to f\, x \in [\![s_2(x)]\!]$.

7.5 Dependent Sum Types

A dependent sum type $\Sigma x{:}s_1.\, s_2$ is an extension of the product type $s_1 \otimes s_2$. Let us assume that s_2 is an expression where x occurs free. If a pair $\langle a, b\rangle$ has the product type $s_1 \otimes s_2$, then a has type s_1 and b has type s_2. If a pair $\langle a, b\rangle$ has the product type $\Sigma x{:}s_1.\, s_2$ and a has type s_1, then b has type $s_2[b/x]$. In other words, the type of b depends on a. Clearly, if s_2 has no free occurrences of x, then $\Sigma x{:}s_1.\, s_2$ reduces to $s_1 \otimes s_2$.

Similar to the dependent type $\Pi x{:}s_1.\, s_2$, the dependent sum type $\Sigma x{:}s_1.\, s_2$ also has binding behavior: it binds x to s_2. Therefore, in the following specification we only define the inhabitant of $\Sigma x{:}s_1.\, s_2$ directly as a notation.

spec DSUM
  Symbol: $\langle\_, \_\rangle$
  Notation:
    $[\![\Sigma x{:}s_1.\, s_2(x)]\!] \equiv \exists x{:}s_1.\, \exists y{:}s_2(x).\, \langle x, y\rangle$
  Axiom:
    (No Confusion):
      $\forall x, x'{:}s_1.\, \forall y{:}s_2(x).\, \forall y'{:}s_2(x').\, \langle x, y\rangle \wedge \langle x', y'\rangle \to \langle x \wedge x', y \wedge y'\rangle$
endspec

Fact 7.28. The following hold:

1. DSUM $\models \forall p{:}(\Sigma x{:}s_1.\, s_2(x)).\, \exists x{:}s_1.\, \exists y{:}s_2(x).\, p = \langle x, y\rangle$.
2. DSUM $\models \forall x{:}s_1.\, \forall y{:}s_2(x).\, \langle x, y\rangle \in [\![\Sigma x{:}s_1.\, s_2(x)]\!]$.

8 Defining Basic Process Algebra as Matching Logic Specifications

8.1 Basic Process Algebra Preliminaries

Process algebra is the field where the behavior of distributed or parallel systems is studied by algebraic means. The most known theories include calculus of communicating systems [13], communicating sequential process [14], π-calculus [15], and algebra of communicating processes [16, 17]. In this paper we consider a simple fragment of algebra of communicating processes called the basic process algebra (BPA). BPA introduces simple operators together with their axioms that enable to describe finite processes. Infinite process can be specified using guarded recursive specifications.

The main ingredients of BPA include:

1. a finite set $Atom$ of atomic actions: $Atom ::= a \mid b \mid c \mid d \mid \cdots$;
2. a set $PTerm$ of process terms denoted $p, q, \ldots$:
$$PTerm ::= Atom \mid PTerm + PTerm \mid PTerm\,; PTerm$$
3. a predicate $p \xrightarrow{u} \surd$ that represents the successful execution of an atomic action $u \in Atom$ of process p;
4. a set of axioms defining the transition relation between process terms:

$$\frac{}{u \xrightarrow{u} \surd} \qquad \frac{x \xrightarrow{u} \surd}{x + y \xrightarrow{u} \surd} \qquad \frac{x \xrightarrow{u} x'}{x + y \xrightarrow{u} x'} \qquad \frac{y \xrightarrow{u} \surd}{x + y \xrightarrow{u} \surd}$$

$$\frac{y \xrightarrow{u} y'}{x + y \xrightarrow{u} y'} \qquad \frac{x \xrightarrow{u} \surd}{x\,; y \xrightarrow{u} y} \qquad \frac{x \xrightarrow{u} x'}{x\,; y \xrightarrow{u} x'\,; y}$$

8.2 Matching Logic Specification of the Basic Process Algebra

spec BPA

Symbol: $Atom, PTerm, \surd, a, b, c, d, \ldots, \_+\_, \_;\_, \bullet, \approx_{\mathrm{BPA}}$

Metavariable: element variables $x, x', y, y' {:} PTerm, u{:}Atom$

Notation:

$\bullet_u\, x \equiv \bullet\, u\, x$

$p + q \equiv \_+\_\ p\, q$

$p\,; q \equiv \_;\_\ p\, q$

$x \approx_{\mathrm{BPA}} y \equiv \langle x, y \rangle \in \approx_{\mathrm{BPA}}$

$$\begin{aligned}\beta(R) \equiv{}& \langle \surd, \surd \rangle \vee \\ & \exists p,q{:}PTerm.\, \langle p, q\rangle \wedge \forall p'{:}PTerm.\, \forall u{:}Atom.\, (p \to \bullet_u\, p') \to \\ &\qquad (\exists q'{:}PTerm.\, q \to \bullet_u\, q' \wedge \langle p', q'\rangle \in R)) \\ &\quad \wedge \forall q'{:}PTerm.\, \forall u{:}Atom.\, (q \to \bullet_u\, q') \to \\ &\qquad (\exists p'{:}PTerm.\, p \to \bullet_u\, p' \wedge \langle p', q'\rangle \in R))\end{aligned}$$

Axiom:

(Sort Name):

$Atom \in [\![Sorts]\!] \qquad PTerm \in [\![Sorts]\!]$

(Function):

$\exists y.\, Atom = y \qquad \exists y.\, PTerm = y$

$\exists y.\, \_+\_ = y \qquad \exists y.\, \_;\_ = y$

$\_+\_ : PTerm \times PTerm \to PTerm$

$\_;\_ : PTerm \times PTerm \to PTerm$

(Domain):

$[\![Atom]\!] = a \vee b \vee c \vee d \vee \cdots$

$[\![PTerm]\!] = \mu X.\, Atom \vee (X + X) \vee (X; X)$

(No Confusion):

$x + y \wedge x' + y' \to (x \wedge x') + (y \wedge y')$

$x\,; y \wedge x'\,; y' \to (x \wedge x')\,; (y \wedge y')$

(Transition):

$\bullet_u [\![PTerm]\!] \subseteq [\![PTerm]\!] \vee \surd$

$\bullet_u \surd = \mu U.\, u \vee U + [\![PTerm]\!] \vee [\![PTerm]\!] + U$

$$\begin{aligned}\bullet_u y = \mu Y.\, (\bullet_u \surd)\,; y \vee Y + [\![PTerm]\!] \vee [\![PTerm]\!] + Y \vee \\ \exists x, x', y'{:}PTerm.\, x\,; y' \wedge y = x'\,; y' \wedge x \in \bullet_u\, x'\end{aligned}$$

(Bisimulation)

$\approx_{\mathrm{BPA}} = \nu R.\, \beta(R)$

$\forall x, y{:}PTerm.\, x \approx_{\mathrm{BPA}} y \leftrightarrow x = y$

endspec

Explanation. The first axiom is equivalent to $\bullet[\![Atom]\!]\,[\![PTerm]\!]) \subseteq [\![PTerm]\!] \vee \surd$ and specifies the signature of the transition relation. The second axiom is the definition of the predicate $\surd$. The third axiom defines the u-predecessors w.r.t. transition relation of a process term y; the correspondence with the axioms is transparent. However, the ML specification is more precise since it defines the exact set of transitions as the least fixpoint. Later it is extended to the greatest fixpoint in order to allow infinite processes. □

Proposition 8.1 (BPA Coinduction Principle)**.**

$$\mathsf{BPA} \models (R \subseteq R' \wedge R' \subseteq \beta(R')) \to (R \subseteq \approx_{\mathrm{BPA}}) \tag{CoindBPA}$$

where $R : PTerm{\otimes}PTerm$.

Explanation. The notation $\beta(R')$ says that R' is a bismulation and $\approx_{\mathrm{BPA}}$ is the largest bisimulation. The proof is similar to that for streams specified with destructors. □

Fact 8.2. The following hold:

1. $\mathsf{BPA} \vDash (\exists p{:}PTerm.\, \langle p + p, p\rangle) \subseteq \approx_{\mathrm{BPA}}$.
2. $\mathsf{BPA} \vDash (\exists p, q{:}PTerm.\, \langle (p + q, q + p\rangle) \subseteq \approx_{\mathrm{BPA}}$.
3. $\mathsf{BPA} \vDash (\exists p, p', q{:}PTerm.\, \langle (p + p')\,;q, p\,; q + p'\,; q\rangle) \subseteq \approx_{\mathrm{BPA}}$.

Explanation. We show the proof trees of Item 1 and leave the rest as exercises. For notational simplicity let us define $\Phi \equiv \exists p{:}PTerm.\, \langle p + p, p\rangle$.

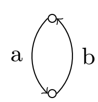
$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathsf{BPA} \vDash \langle p' + p', p'\rangle \in \Phi}{\mathsf{BPA} \vDash (p \to \bullet_u p' \wedge (p+p) \to \bullet_u (p'+p')) \to p \to \bullet_u p' \wedge \langle p' + p', p'\rangle \in \Phi}\,\mathrm{FOL} \qquad \cdots}{\begin{array}{l}\mathsf{BPA} \vDash \forall p'{:}PTerm.\, \forall u{:}Atom.\, ((p+p) \to \bullet_u\, p') \to (\exists q'{:}PTerm.\, p \to \bullet_u\, q' \wedge \langle p', q'\rangle \in \Phi)) \\ \quad \wedge\, \forall q'{:}PTerm.\, \forall u{:}Atom.\, (p \to \bullet_u\, q') \to (\exists p'{:}PTerm.\, (p+p) \to \bullet_u\, p' \wedge \langle p', q'\rangle \in \Phi))\end{array}}\,\bullet_u p'}{\mathsf{BPA} \vDash \langle p + p, p\rangle \to \beta(\Phi)}\,\mathrm{FOL}}{\mathsf{BPA} \vDash (\exists p{:}PTerm.\, \langle p + p, p\rangle) \to \beta(\Phi)}\,\exists\text{-}\mathrm{Gen}}{\mathsf{BPA} \vDash \Phi \subseteq \approx_{\mathrm{BPA}}}\,\mathrm{KT}$$

□

8.3 Guarded Recursive Specifications

The infinite processed are specified using guarded recursive specifications. Here is a very simple example:

$$x = a\,;y \qquad y = b\,;x$$



First we extend the definition of terms to describe infinite processes: $[\![PTerm]\!] = \nu X.\, Atom \vee X + X \vee X; X$. Then the two processes are specified together using the product sort $PTerm{\otimes}PTerm$:

$$\langle p_x, p_y\rangle = \nu P{:}PTerm{\otimes}PTerm.\, \langle a\,; \pi_2(P), b\,; \pi_1(P)\rangle$$

9 Functors and (Co)Monads as Matching Logic Specifications

In this section we show how the higher-order reasoning in category theory can be internalized in ML. We give specifications for functors, monads, and comonads as they are defined in functional languages, like Haskell (see, e.g., [18, 19]).

9.1 Functors

We first enrich $[\![Sorts]\!]$ with a "category structure":

spec CAT
  Symbol: id , $\circ$
  Metavariable: element variables $s, s_1, s_2, s_3{:}Sorts$
  Notation:
    $g \circ h \equiv \circ\, g\, h$
  Axiom:
    (Function) :
      $\exists y.\, id = y$
      $\exists y.\, \circ = y$
    (Identity and Composition Laws) :
      $(id\ s){:}s \mathrel{\ominus\!\!\!\!\rightarrow} s$
      $\forall x{:}s.\, (id\ s)\, x = x$
      $\forall g_1{:}s_1 \mathrel{\ominus\!\!\!\!\rightarrow} s_2.\, \forall g_2{:}s_2 \mathrel{\ominus\!\!\!\!\rightarrow} s_3.\, (g_2 \circ g_1){:}s_1 \mathrel{\ominus\!\!\!\!\rightarrow} s_3$
      $\forall x{:}s_1.\, \forall g_1{:}s_1 \mathrel{\ominus\!\!\!\!\rightarrow} s_2.\, \forall g_2{:}s_2 \mathrel{\ominus\!\!\!\!\rightarrow} s_3.\, (g_2 \circ g_1)\, x = g_2\, (g_1\, x)$
endspec

Explanation. The objects of the category are given by sorts, and the arrows by the inhabitants of function sorts $s \mathrel{\ominus\!\!\!\!\rightarrow} s'$. The axioms of the category are self-explaining. Recall that $g : s_1 \mathrel{\ominus\!\!\!\!\rightarrow} s_2 \equiv g \in [\![s_1 \mathrel{\ominus\!\!\!\!\rightarrow} s_2]\!]$. □

Fact 9.1.

$$\mathsf{CAT} \models \forall g{:}s_1 \mathrel{\ominus\!\!\!\!\rightarrow} s_2.\, (g \circ (id\ s_1)) =^{s_1}_{ext} g$$

$$\mathsf{CAT} \models \forall g{:}s_1 \mathrel{\ominus\!\!\!\!\rightarrow} s_2.\, ((id\ s_2) \circ g) =^{s_1}_{ext} g \qquad (\circ\textsc{IdL})$$

$$\mathsf{CAT} \models \forall g_1{:}s_1 \mathrel{\ominus\!\!\!\!\rightarrow} s_2\, .\forall g_2{:}s_2 \mathrel{\ominus\!\!\!\!\rightarrow} s_3\, .\forall g_3{:}s_3 \mathrel{\ominus\!\!\!\!\rightarrow} s_4\, .(g_3 \circ (g_2 \circ g_1)) =^{s_1}_{ext} ((g_3 \circ g_2) \circ g_1) \qquad (\circ\textsc{Assoc})$$

The explanation for the above fact is quite simple and is left as an exercise to the reader.

Matching Logic specification of a functor It is given by the mean of two symbols f and map as follows:

spec FNCTR
  Import: CAT
  Symbol: f, map
  Metavariable: element variables $s, s_1, s_2{:}Sorts$
  Axiom:
    (Function) :
      $\exists y.\, f = y$
      $\exists y.\, map = y$
    (Functor Laws) :
      $f : Sorts \to Sorts$
      $\forall g{:}s_1 \mathrel{\ominus\!\!\!\!\rightarrow} s_2.\, (map\ g){:}(f\ s_1) \mathrel{\ominus\!\!\!\!\rightarrow} (f\ s_2)$
      $(map\ (id\ s)) = id\ s$
      $\forall h{:}s_1 \mathrel{\ominus\!\!\!\!\rightarrow} s_2.\, \forall g{:}s_2 \mathrel{\ominus\!\!\!\!\rightarrow} s_3.\, map\ (g \circ h) =^{m\, s_1}_{ext} (map\ g) \circ (map\ h)$ (MDist)
endspec

Explanation. The objects mapping is given by the first (Functor Laws) axiom and the arrows mapping is given the second one. The last two axioms say that a functor preserves the identities and the composition. □

9.2 Monads

9.2.1 Monads Categorically

Recall that in the category theory, a monad consists of a functor (m, map) and two natural transformations: $\mu : m^2 \to m$ (*join*, *multiplication*), and $\eta : \mathbf{1}_m \to m$ (*unit*) satisfying the following equations:

$$\mu \circ (\eta\, m) = \mathbf{1}_m$$
$$\mu \circ (m\, \eta) = \mathbf{1}_m$$
$$\mu \circ (m\, \mu) = \mu \circ (\mu m)$$

where $\mathbf{1}_m : m \to m$ is the identity natural transformation. The natural transformation μ associates an arrow $\mu_s : m^2\, s \to m\, s$ for each $s{:}Sorts$. Similarly, η associates an arrow $\eta_s : s \to m\, s$ for each $s{:}Sorts$. Not that the above equalities express the commutativity of diagrams in term of the category theory.

Matching Logic Specification

spec MONAD
  Import: FNCTR
  Symbol: μ, η
  Metavariable: element variables $s, s_1, s_2{:}Sorts$
  Notation:
    $\mu_s \equiv \mu\ s$
  Axiom:
    (Natural Transformations):
      $\mu_s{:}((m \circ m)\, s) \ominus\!\!\!\!\rightarrow (m\, s)$
      $\nu_s{:}s \ominus\!\!\!\!\rightarrow (m\, s)$
      $\forall\, g{:}s_1 \ominus\!\!\!\!\rightarrow s_2.\ \eta_{s_2} \circ g =^{s_1}_{ext} (map\ g) \circ \eta_{s_1}$ (ηNt)
      $\forall\, g{:}s_1 \ominus\!\!\!\!\rightarrow s_2.\ (\mu_{s_2} \circ (map\ map\ g)) =^{m\, m\, s_1}_{ext} (map\ g) \circ \mu_{s_1}$ (μNt)
    (Diagram Commutativity)
      $\mu_s \circ \eta_{(m\, s)} =^{s}_{ext} id_{(m\, s)}$ (ηIdR)
      $\mu_s \circ (map\, \eta_s) =^{s}_{ext} id_{(m\, s)}$ (ηIdL)
      $\mu_{m\, s} =^{m\, m\, m\, s}_{ext} map\, \mu_s$ (μmComm)
endspec

Explanation. We preferred to use the axiom (μmComm) instead of (μAssoc) (see below), which is proved as semantic consequence. We use the axiom (1) later. □

Fact 9.2.

$$\text{MONAD} \models \forall s{:}Sorts.\ \mu_s \circ (map\, \mu_s) =^{m\, m\, s}_{ext} \mu_s \circ \mu_{(m\, s)} \qquad (\mu\text{Assoc})$$
$$\text{MONAD} \models \forall\, g{:}s_1 \ominus\!\!\!\!\rightarrow m\, s_2.\ map\, (\mu_{s_2} \circ (map\ g)) =^{m\, m\, s_1}_{ext} (map\ g) \circ \mu_{s_1} \qquad (\mu\text{Nt2})$$

Explanation. (μAssoc) follows by applying (1). (μNt2) is explained as follows:

$$\begin{aligned} map\, (\mu_{s_2} \circ (map\ g)) &=^{m\, m\, s_1}_{ext} (map\, \mu_{s_2}) \circ (map\ map\ g) && \text{(by (MDist))} \\ &=^{m\, m\, s_1}_{ext} (\mu_{m\, s_2}) \circ (map\ map\ g) && \text{(by (}\mu m\text{Comm))} \\ &=^{m\, m\, s_1}_{ext} (map\ g) \circ \mu_{s_1} && \text{(by (}\mu\text{Nt))} \end{aligned}$$

□

9.2.2 Monads in Functional Programming Languages

In programming languages and semantics the Kleisli alternative definition for monads is used (see, e.g., [20]). Roughly speaking, this consists of considering a symbol *bind* (denoted also by ≫=) instead of μ defined by the following axiom, which we add to MONAD:

$$\forall\, g{:}s_1 \ominus m\, s_2.\; bind\; g =^{m\, s_1}_{ext} \mu_{s_2} \circ map\; g$$

A common name for the unit η in programming languages is that of *return*. We prove now the properties of *bind*, which characterize it in Kleisli categories and programming languages:

Fact 9.3. The following hold:

1. $\mathsf{MONAD} \models (bind\; g) \circ \eta_{s_1} =^{s_1}_{ext} g$
2. $\mathsf{MONAD} \models bind\; \eta_s =^{m\, s}_{ext} id_{m\, s}$
3. $\mathsf{MONAD} \models \forall\, g{:}s_1 \ominus m\, s_2.\, \forall\, h{:}s_2 \ominus m\, s_3.\; bind((bind\; h) \circ g) =^{m\, s_3}_{ext} bind\; h \circ bind\; g$

Explanation. We have the following reasoning.
(Item 1).

$$\begin{aligned}
(bind\; g) \circ \eta_{s_1} &=^{m\, s_1}_{ext} (\mu_{s_2} \circ (map\; g)) \circ \eta_{s_1} && \text{(by definition of } bind\text{)}\\
&=^{m\, s_1}_{ext} \mu_{s_2} \circ ((map\; g) \circ \eta_{s_1}) && \text{(by } (\circ\textsc{Assoc})\text{)}\\
&=^{m\, s_1}_{ext} \mu_{s_2} \circ (\eta_{m\, s_2} \circ g) && \text{(by } (\eta\textsc{Nt})\text{)}\\
&=^{m\, s_1}_{ext} (\mu_{s_2} \circ \eta_{m\, s_2}) \circ g && \text{(by } (\circ\textsc{Assoc})\text{)}\\
&=^{m\, s_1}_{ext} id_{m\, s_2} \circ g && \text{(by } (\eta\textsc{Idr})\text{)}\\
&=^{m\, s_1}_{ext} g && \text{(by } (\circ\textsc{Idl})\text{)}
\end{aligned}$$

(Item 2).

$$\begin{aligned}
bind\; \eta_s &=^{m\, s}_{ext} \mu_s \circ (map\; \eta_s) && \text{(by definition of } bind\text{)}\\
&=^{m\, s}_{ext} id_{m\, s} && \text{(by } (\eta\textsc{Idl})\text{)}
\end{aligned}$$

(Item 3).

$$\begin{aligned}
&bind((bind\; h) \circ g)\\
&\quad =^{m\, s_3}_{ext} bind((\mu_{s_3} \circ (map\; h)) \circ g) && \text{(by definition of } bind\text{)}\\
&\quad =^{m\, s_1}_{ext} \mu_{s_3} \circ map\; ((\mu_{s_3} \circ (map\; h)) \circ g) && \text{(by definition of } bind\text{)}\\
&\quad =^{m\, s_1}_{ext} \mu_{s_3} \circ map\; (\mu_{s_3} \circ (map\; h)) \circ (map\; g) && \text{(by (MD}\textsc{ist}\text{),(}\circ\textsc{Assoc}\text{))}\\
&\quad =^{m\, s_1}_{ext} \mu_{s_3} \circ ((map\; h) \circ \mu_{s_2}) \circ (map\; g) && \text{(by } (\mu\textsc{Nt}2)\text{)}\\
&\quad =^{m\, s_1}_{ext} (\mu_{s_3} \circ (map\; h)) \circ (\mu_{s_2} \circ (map\; g)) && \text{(by } (\circ\textsc{Assoc})\text{)}\\
&\quad =^{m\, s_1}_{ext} bind\; h \circ bind\; g && \text{(by definition of } bind\text{)}
\end{aligned}$$

□

9.3 Comonads

9.3.1 Comonads Categorically

In the category theory, the notion of comonad is defined as the dual of that of monad. Consequently, a comand consists of a functor (w, map) and two natural transformations: $\delta : w \to w^2$ (*duplication*, *comultiplication*), and $\varepsilon : w \to \mathbf{1}_w$ satisfying the following equations:

$$(\varepsilon\, w) \circ \delta = \mathbf{1}_w$$

$$(w\,\varepsilon) \circ \delta = \mathbf{1}_w$$
$$w\,\delta \circ \delta = \delta\,w \circ \delta$$

where $\mathbf{1}_w : m \to m$ is the identity natural transformation. The natural transformation δ associates an arrow $\delta_s : w\,s \to w^2\,s$ for each $s{:}Sorts$. Similarly, ε associates an arrow $\varepsilon_s : w\,s \to s$ for each $s{:}Sorts$. Not that the above equalities express the commutativity of diagrams in term of the category theory.

Matching Logic Specification

spec COMONAD
  Import: FNCTR
  Symbol: δ, ε
  Notation:
    $\delta_s \equiv \delta\,s$
    $\varepsilon_s \equiv \varepsilon\,s$
  Axiom:
    (Natural Transformations) :
      $\forall s{:}Sorts.\, \delta_s{:}(w \circ w\,s) \ominus\!\!\!\rightarrow (w\,s)$
      $\forall s{:}Sorts.\, \varepsilon_s{:}(w\,s) \ominus\!\!\!\rightarrow s$
      $\forall\, g{:}s_1 \ominus\!\!\!\rightarrow s_2.\, g \circ \varepsilon_{s_1} =^{w\,s_1}_{ext} \varepsilon_{s_2} \circ (map\,g)$ (εNt)
      $\forall\, g{:}s_1 \ominus\!\!\!\rightarrow s_2.\, \delta_{s_2} \circ (map\,g) =^{w\,w\,s_1}_{ext} (map\,map\,g) \circ \delta_{s_1}$ (δNt)
    (Diagram Commutativity)
      $\forall s{:}Sorts.\, \varepsilon_{w\,s} \circ \delta_s =^{s}_{ext} id_{w\,s}$ (εIdR)
      $\forall s{:}Sorts.\, (map\,\varepsilon_s) \circ \delta_s =^{s}_{ext} id_{w\,s}$ (εIdL)
      $\forall s{:}Sorts.\, map\,\delta_s =^{w\,s}_{ext} \delta_{w\,s}$ ($w\delta$Comm)
endspec

Fact 9.4.

$$\mathsf{COMONAD} \models \forall s{:}Sorts.\, (map\,\delta_s) \circ \delta_s =^{w\,s}_{ext} \delta_{w\,s} \circ \delta_s \qquad (\delta\text{Assoc})$$
$$\mathsf{COMONAD} \models \forall\, g{:}w\,s_1 \ominus\!\!\!\rightarrow s_2.\, map\,((map\,g) \circ \delta_{s_1}) =^{w\,w\,s_1}_{ext} \delta_{s_2} \circ (map\,g) \qquad (\delta\text{Nt2})$$

Explanation. Similar to that of Fact 9.2. □

9.3.2 Comonads in Functional Programming Languages

Similar to monads, a comonad is defined in programming languages using a symbol *cobind* (*extend*) defined by the following axiom, which we add to COMONAD:

$$\forall\, g{:}w\,s_1 \ominus\!\!\!\rightarrow s_2.\, cobind\,g =^{w\,s_2}_{ext} map\,g \circ \delta_{s_1}$$

We prove now the properties of *cobind*, which characterize it in coKleisli categories and programming languages:

Fact 9.5. The following hold:

1. COMONAD $\models \varepsilon_{s_2} \circ (cobind\,g) =^{w\,s_1}_{ext} g$
2. COMONAD $\models cobind\,\varepsilon_s =^{m\,s}_{ext} id_{w\,s}$
3. COMONAD $\models \forall\, g{:}w\,s_1 \ominus\!\!\!\rightarrow s_2. \forall\, h{:}w\,s_2 \ominus\!\!\!\rightarrow s_3.\, cobind(h \circ (cobind\,g)) =^{m\,s_1}_{ext} cobind\,h \circ cobind\,g$

Explanation. Similar to that of Fact 9.3. □

10 Conclusion

In this paper we gave an example-driven, yet comprehensive introduction to matching logic. We showed how to use matching logic specifications to capture various mathematical domains and data types, and we proposed matching logic notations to define domain-specific languages. We explained technical details when writing matching logic specifications and reasoning about matching logic semantics. In particular we discussed how to carry out inductive and coinductive reasoning using matching logic.

References

[1] G. Roşu, Matching logic, Logical Methods in Computer Science 13 (4) (2017) 1–61. doi:10.23638/LMCS-13(4:28)2017.

[2] X. Chen, G. Rosu, Matching μ-logic, in: Proceedings of the 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2019), IEEE, Vancouver, Canada, 2019, pp. 1–13. doi:10.1109/LICS.2019.8785675.

[3] X. Chen, G. Roşu, Applicative matching logic, Tech. Rep. http://hdl.handle.net/2142/104616, University of Illinois at Urbana-Champaign (July 2019).

[4] A. Arusoaie, D. Lucanu, Unification in matching logic, in: Proceedings of the 3rd World Congress on Formal Methods (FM 2019), 2019, pp. 502–518. doi:10.1007/978-3-030-30942-8_30.

[5] X. Chen, G. Roşu, A general approach to define binders using matching logic, Tech. Rep. http://hdl.handle.net/2142/106608, University of Illinois at Urbana-Champaign (2020).

[6] X. Chen, D. Lucanu, G. Roşu, Initial algebra semantics in matching logic, Tech. Rep. http://hdl.handle.net/2142/107781, University of Illinois at Urbana-Champaign and Alexandru Ioan Cuza University (2020).

[7] A. Tarski, A lattice-theoretical fixpoint theorem and its applications., Pacific J. Math. 5 (2) (1955) 285–309.

[8] J. A. Goguen, An initial algebra approach to the specification, correctness and implementation of abstract data types, IBM Research Report 6487 (1976).

[9] B. Stroustrup, Concepts: the future of generic programming or how to design good concepts and use them well (2017).
URL https://www.stroustrup.com/good_concepts.pdf

[10] A. Sutton, Defining concepts, Overload Journal (131) (2016).
URL https://accu.org/index.php/journals/2198

[11] M. Niqui, J. J. M. M. Rutten, Stream processing coalgebraically, Sci. Comput. Program. 78 (11) (2013) 2192–2215. doi:10.1016/j.scico.2012.07.013.

[12] G. Roşu, D. Lucanu, Circular coinduction – a proof theoretical foundation, in: Proceedings of the 3rd International Conference on Algebra and Coalgebra in Computer Science (CALCO 2009), Springer, Udine, Italy, 2009, pp. 127–144. doi:10.1007/978-3-642-03741-2_10.

[13] R. Milner (Ed.), A calculus of communicating systems, Springer, 1980. doi:10.1007/3-540-10235-3.

[14] C. A. R. Hoare, Communicating sequential processes, Prentice-Hall, 1985.

[15] R. Milner, Communicating and mobile systems - the Pi-calculus, Cambridge University Press, 1999.

[16] J. C. M. Baeten, W. P. Weijland, Process algebra, Cambridge University Press, 1990.

[17] W. Fokkink, Introduction to process algebra, Springer, 2000. doi:10.1007/978-3-662-04293-9.

[18] T. Uustalu, V. Vene, Comonadic notions of computation, Electr. Notes Theor. Comput. Sci. 203 (5) (2008) 263–284. doi:10.1016/j.entcs.2008.05.029.

[19] D. Orchard, Should I use a monad or a comonad?, draft work (2012).
URL https://www.cs.kent.ac.uk/people/staff/dao7/drafts/monad-or-comonad-orchard11-draft.pdf

[20] Haskell monads, last visit December 2019.
URL https://wiki.haskell.org/Monad