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`

May 25, 2021

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 of 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 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 future tools development 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 *functional variant*, which has a minimal presentation among the others. We refer to this variant as *matching logic* throughout the paper, abbreviating it as ML.

The key concept of ML is its *patterns*. Patterns are ML formulas, which are built from variables, constant symbols, a binary *application* construct, standard FOL constructs $\bot$, $\to$, $\exists$, and a least fixpoint construct μ. In terms of semantics, patterns are interpreted as the 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 then a disjunctive[1] 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 various examples in Sections 3 to 9.

We can use patterns to *constrain models*, by enforcing the models to match a set of given patterns, called *axioms*. This set of axioms yields a *specification*, also called a logical theory. In this paper we will define

[*]This technical report is published in the Journal of Logical and Algebraic Methods in Programming, vol. 120, which is available at `https://doi.org/10.1016/j.jlamp.2021.100638`.

[1]Disjunction can be defined as syntactic sugar in the usual way. See Section 2.1 for more details on syntactic sugar definitions.

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 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 explain how patterns are interpreted in ML models.
– In Section 5 we review the Hilbert-style proof system of ML and its soundness theorem.
– 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, 3] for full technical details. The ML variant that we introduce in this paper is called the functional variant and is firstly proposed in [3], but there fixpoints are not considered. Therefore, the syntax and semantics definitions that we will give in this section are an extension of the work in [3] by fixpoints.

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.

Intuitively, element variables are FOL-style variables that range over the individual elements in the models (Section 4) while set variables range over the sets of elements. Symbols are like set variables in that they also represent sets of elements, but unlike set variables, the interpretations of symbols are directly given and fixed by the underlying models.

Definition 2.2. Given $\mathbb{\Sigma} = (EV, SV, \Sigma)$, the set $\textsc{Pattern}(\mathbb{\Sigma})$ of $\mathbb{\Sigma}$-*patterns* (or simply *patterns*) is inductively 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 $FV(\varphi) \subseteq EV \cup SV$ to denote the set of (element and set) free variables in φ, i.e., variables that are not in the scope of any $\exists$ or μ binders. 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 constructs as syntactic sugar in the usual way and assume the standard precedence among them:

$$\begin{aligned} \neg\varphi &\equiv \varphi \to \bot & \varphi_1 \vee \varphi_2 &\equiv \neg\varphi_1 \to \varphi_2 \\ \varphi_1 \wedge \varphi_2 &\equiv \neg(\neg\varphi_1 \vee \neg\varphi_2) & \top &\equiv \neg\bot \\ \varphi_1 \leftrightarrow \varphi_2 &\equiv (\varphi_1 \to \varphi_2) \wedge (\varphi_2 \to \varphi_1) & \forall x.\, \varphi &\equiv \neg\exists x.\, \neg\varphi \\ \nu X.\, \varphi &\equiv \neg\mu X.\, \neg\varphi[\neg X/X] \end{aligned}$$

The greatest fixpoint pattern $\nu X.\, \varphi$ is defined from the least fixpoint in the usual way; see, e.g., [**?**, Section 3.3].

We point out that the above are syntactic sugar, i.e., notations, so they do not extend the logic and can be eliminated entirely. It is also important to note that these constructs are not symbols. For example, $\neg\varphi$ is not the (matching logic) symbol $\neg$ applied to φ, because the semantics of symbols (explained in Section 2.2) do not apply to these logical constructs. As we will see in Definition 2.4, the result of a symbol applying to $\bot$ is always $\bot$, in terms of the semantics. If $\neg$ were defined as a symbol, this would result in $\neg\bot$ being equal to $\bot$, which is not what we want.

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 the *set* of elements that *match* it. Intuitively, the pattern $\bot$ (called bottom) is matched by no elements, while $\top$ (called top) is matched by all elements. Conjunction $\varphi_1 \wedge \varphi_2$ is matched by the elements that match both φ_1 and φ_2, disjunction $\varphi_1 \vee \varphi_2$ by the elements that match φ_1 or φ_2, negation $\neg\varphi$ by the elements that do not match φ, and implication $\varphi_1 \to \varphi_2$ by all elements a such that if a matches φ_1 then a matches φ_2. Element variable x is matched by the element to which x evaluates (see Definition 2.4). Set variable X is matched by the set of elements to which X evaluates; this set can be empty, or total, or any subset of the carrier set. Application $\varphi_1\, \varphi_2$ is used to build structures, relations, and propositions, whose actual semantics depends on φ_1. For example, if $\varphi_1 \equiv f$ is a function symbol, then $f\, x$ denotes the (FOL-like) term $f(x)$. By currying the application, we get $f\, x_1\, \ldots\, x_n$ that represents the term $f(x_1, \ldots, x_n)$. Similarly, if $\varphi_1 \equiv p$ is a predicate symbol, then $f\, x$ denote the proposition $p(x)$, stating that x holds for p. Quantification $\exists x.\, \varphi$ is matched by the elements that match φ for *some* valuation of x; that is, it abstracts away the irrelevant part x from the matched part φ. Least fixpoint $\mu X.\, \varphi$ is matched by the smallest set X of elements that satisfies the equation $X = \varphi$ (this is interesting when X occurs in φ).

Below, we first define ML models.

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 is a carrier set, required to be nonempty;
2. $\_\bullet\_ : M \times M \to \mathcal{P}(M)$ 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, called the *interpretation of* σ *in* M for each $\sigma \in \Sigma$.

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

Compared to FOL, ML enforces a powerset interpretation for both application and symbols. In FOL models, application as a binary function is interpreted by a function from $M \times M$ to M and a constant function symbol c is interpreted as an element $M_c \in M$ in the carrier set. However, ML requires application and symbols to return a set of elements. Note that the set M is isomorphic to the set of singleton sets in $\mathcal{P}(M)$. Therefore, the FOL-style interpretation can be regarded as a special instance of the ML powerset interpretation where all returning sets are singleton sets.

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 $\Sigma = (EV, SV, \Sigma)$ 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 a *pattern interpretation* $|\_|_\rho \colon \text{PATTERN} \to \mathcal{P}(M)$ 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 Proposition 2.6). 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 [6] (see Proposition 2.7).

Example 2.5. As an example, we show that $|\mu X.\, X|_\rho = \emptyset$. By definition, $|\mu X.\, X|_\rho = \mu\mathcal{F}^\rho_{X,X}$, where $\mathcal{F}^\rho_{X,X}(A) = A$ for all $A \subseteq M$. In other words, any set A is a fixpoint of $\mathcal{F}^\rho_{X,X}$. Since $\mu\mathcal{F}^\rho_{X,X}$ denotes the least fixpoint, it equals to the empty set $\emptyset$.

In Proposition 2.7, we give a more "constructive" definition of the fixpoint $\mu\mathcal{F}^\rho X, \varphi$ based on the Knaster-Tarski fixpoint theorem [6].

Proposition 2.6. *The function $\mathcal{F}^\rho_{X,\varphi}$ 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 $A \subseteq B$.*

Proposition 2.7. *The function $\mathcal{F}^\rho_{X,\varphi}$ has a unique least fixpoint and a unique greatest fixpoint, both given as below:*

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

As one expects, the interpretation of a pattern φ only depends on the valuations of variables that are free in φ.

Proposition 2.8. *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 $|\varphi|_{\rho_1} = |\varphi|_{\rho_2}$.*

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, ML patterns are interpreted as subsets of M, different from how FOL formulas are interpreted as either true or false. A natural question is how to represent in ML propositions or statements, whose semantics can take only two values: true or false. The answer is that we can use two special sets, the total set M and the empty set $\emptyset$, to represent true and false. Since we require the carrier set M to be nonempty, the two sets M and $\emptyset$ are never the same. Formally, given a model 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. A predicate pattern can be regarded as a proposition or a 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 throughout the paper.

Next, we define semantic validity in ML.

Definition 2.9. 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.

The above definition of ML validity should be expected. Note that in defining $M \vDash \varphi$, we require $|\varphi|_\rho = M$ for all valuations ρ. In other words, all elements in M match the pattern φ. It is a natural definition since we use the total set M to denote the logical truth. Therefore, a pattern holds in a model if it evaluates to "true", i.e., the total set M, in the model.

For a pattern set Γ, we say that M validates Γ if and only if all patterns in Γ hold in M, that is, all of them evaluate to true (the total set) in M. Thus, we often call the patterns in Γ *axioms* and call Γ the axiom set. A *specification* is a pair of a signature and an axiom set of patterns of that signature. Formally,

Definition 2.10. A matching logic *specification* SPEC is a tuple (EV, SV, Σ, Γ), 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 and $M \vDash \mathsf{SPEC}$ 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. Therefore, we only need to specify the symbol set Σ and the axiom set Γ. In Section 3, we will define several example specifications using a human-readable, semi-formal syntax.

3 Specification Examples: Important Mathematical Instruments

We define several important mathematical instruments such as equality, membership, sorts, functions, predicates, and constructors, as ML specifications. We will start with equality and membership, both of which are defined using the *definedness symbol*. Then, we will axiomatically define sorts in unsorted ML and show that the sorted functions and predicates can be defined as ML symbols in a direct and easy way. Our last example is to axiomatize a set of functions as the *constructors* of a given sort, which requires us to capture the principle of structural induction over constructor terms using the fixpoint patterns in ML.

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

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

Intuitively, (DEFINEDNESS) states that all elements are defined. In other words, any element variable x is a defined pattern, because according to ML semantics (Definition 2.4), x is always interpreted as a singleton set and thus is nonempty.

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

```
spec  DEFINEDNESS
   Symbol:  $\lceil \_ \rceil$
   Metavariable:   a pattern  $\varphi$
   Notation:
      $\lceil \varphi \rceil \equiv \lceil \_ \rceil\, \varphi$
   Axiom:
      (DEFINEDNESS)  $\forall x.\, \lceil x \rceil$
endspec
```

Here, keyword Symbol enumerates the symbols declared in the specification. Keyword Metavariable enumerates the metavariables that are used in the specification. In this specification, the metavariable φ is used only in the notation definition (explained below), where we use φ to range over all ML patterns. Later, we will see specifications that use metavariables in axiom definitions. Keyword Notation introduces notations (syntactic sugar). Keyword Axiom lists all axioms (schemas). We give names to important axioms for better readability, like (DEFINEDNESS), and we omit the metavariable declarations when they can be easily inferred. Then, 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)  $\forall x.\, \lceil x \rceil$
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 $|\varphi|_\rho = \emptyset$.*

Therefore, $\lceil \varphi \rceil$ is the predicate that holds whenever φ is defined, i.e., $\lceil \varphi \rceil$ evaluates to true (the total set M) if φ is defined and it evaluates to false (the empty set $\emptyset$) otherwise.

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

Intuitively, $\lfloor \varphi \rfloor$ is the dual predicate of $\lceil \varphi \rceil$. While $\lceil \varphi \rceil$ checks non-emptiness of the interpretation of φ, $\lfloor \varphi \rfloor$ checks totality, i.e., whether φ evaluates to the total set or not. If so, $\lfloor \varphi \rfloor$ holds. If not, it fails. Therefore, $\lfloor \varphi \rfloor$ is called *totality.* Equality $\varphi_1 = \varphi_2$ is defined using totality and $\varphi_1 \leftrightarrow \varphi_2$. Following Definition 2.4, one can show that $\varphi_1 \leftrightarrow \varphi_2$ represents the complement of the symmetric difference between φ_1 and φ_2. In other words, $\varphi_1 \leftrightarrow \varphi_2$ is matched by the elements a such that a matches φ_1 iff it matches φ_2. Then, φ_1 and φ_2 are equal iff $\varphi_1 \leftrightarrow \varphi_2$ is matched by all elements, and thus we define $\varphi_1 = \varphi_2 \equiv \lfloor \varphi_1 \leftrightarrow \varphi_2 \rfloor$. Membership $x \in \varphi$ is defined for an element variable x and a pattern φ, meaning that the (unique) element matching x also matches φ. In other words, $x \wedge \varphi$ does not evaluate to the empty set. Set inclusion $\varphi_1 \subseteq \varphi_2$ is defined like equality, where we use $\varphi_1 \to \varphi_2$ instead of $\varphi_1 \leftrightarrow \varphi_2$. Therefore, φ_1 is included in φ_2 iff all elements matching φ_1 also match φ_2, i.e., iff $\varphi_1 \to \varphi_2$ is matched by all elements.

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

Proposition 3.3. *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 $\llbracket \_ \rrbracket$, called the *inhabitant symbol*, to build the *inhabitant pattern* $\llbracket \_ \rrbracket\, s$, often written as $\llbracket s \rrbracket$, 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
  Symbol: $\llbracket \_ \rrbracket, \mathit{Sorts}$
  Notation:
    $\llbracket s \rrbracket \equiv \llbracket \_ \rrbracket\, s$
    $\neg_s \varphi \equiv (\neg \varphi) \wedge \llbracket s \rrbracket$
    $\forall x{:}s.\, \varphi \equiv \forall x.\, (x \in \llbracket s \rrbracket) \to \varphi$
    $\exists x{:}s.\, \varphi \equiv \exists x.\, (x \in \llbracket s \rrbracket) \wedge \varphi$
endspec

Here, keyword Import imports all the symbols, notations, and axioms defined in specification DEFINEDNESS. Symbol $\llbracket \_ \rrbracket$ is 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 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 $s_1, s_2, \ldots$, $F = \{F_{s_1 \cdots s_n, s}\}_{s_1, \ldots, s_n, s \in S}$ is a family set of *many-sorted functions* denoted $f \in F_{s_1 \cdots s_n, s}$, and $\Pi = \{\Pi_{s_1 \cdots s_n}\}_{s_1, \ldots, s_n \in S}$ 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 $s_1, \ldots, s_n$ 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 $\llbracket s \rrbracket$. 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) $(s \in \llbracket Sorts \rrbracket) \wedge (\exists z.\, s = z)$
(Nonempty Inhabitant) $\llbracket s \rrbracket \neq \bot$
(Function) $\forall x_1{:}s_1 \ldots \forall x_n{:}s_n.\, \exists y{:}s.\, f\, x_1 \cdots x_n = y$
(Predicate) $\forall x_1{:}s_1 \ldots \forall x_n{:}s_n.\, \pi\, x_1 \cdots x_n = \top \vee \pi\, x_1 \cdots x_n = \bot$
endspec

We explain the above specification. 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 inhabitants 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 $x_1, \ldots, x_n$ have sorts $s_1, \ldots, s_n$, respectively. In other words, f is a *many-sorted function* from $\llbracket s_1 \rrbracket \times \cdots \times \llbracket s_n \rrbracket$ to $\llbracket s \rrbracket$. Similarly, (Predicate) specifies that π is a *many-sorted predicate* on $s_1, \ldots, s_n$, 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 $f \colon \epsilon \to s$.

Therefore, many-sorted functions and predicates can be uniformly represented using ML symbols and their behaviors can be defined using ML axioms (Function) and (Predicate), respectively. This demonstrates the philosophy that we pursue with ML, where we adopt a minimalist design for the syntax and semantics of ML and only include the absolute necessary constructs. The other more complex constructs and concepts such as sorts and functions can be defined using axioms and their properties can be reasoned about using the proof system (Section 5). This makes ML highly flexible because it does not stick to any particular formalisms or existing approaches.

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:

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

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:

$$\text{(Subsorting)} \quad \llbracket s_1 \rrbracket \subseteq \llbracket s_2 \rrbracket$$

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:

$$\text{(Sorted Symbol)} \quad \sigma\, \llbracket s_1 \rrbracket \cdots \llbracket s_n \rrbracket \subseteq \llbracket s \rrbracket$$

Notation 3.4. Let s be a sort and M be a model, the interpretation $|[\![s]\!]|$ is the carrier set of s in M. For notational simplicity, we write $[\![s]\!]_M$ as an abbreviation of $|[\![s]\!]|$ to denote the carrier set.

3.3 Constructors and 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 [7].[2] Specifically, let *Term* be a distinguished sort for terms and $C = \{c_1, c_2, \dots\}$ be a set of *constructors*. For each c_i, we associate an arity $\mathit{arity}(c_i) \in \mathbb{N}$.[3] We define the following ML specification:

spec CONSTRUCTORS$\{C\}$
  Import: MANYSORTED$\{\{\mathit{Term}\}, C, \emptyset\}$
  Metavariable: $c, d \in C$
  Axiom:
    (No Confusion) where $n = \mathit{arity}(c), m = \mathit{arity}(d)$
      $\forall x_1{:}\mathit{Term} \dots \forall x_n{:}\mathit{Term}. \forall y_1{:}\mathit{Term} \dots \forall y_m{:}\mathit{Term}.$
        $c\, x_1 \cdots x_n \neq d\, y_1 \cdots y_m$
      $\forall x_1{:}\mathit{Term} \dots \forall x_n{:}\mathit{Term}. \forall y_1{:}\mathit{Term} \dots \forall y_n{:}\mathit{Term}.$
        $c\, x_1 \cdots x_n = c\, y_1 \cdots y_n \to x_1 = y_1 \wedge \cdots \wedge x_n = y_n$
    (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 constructors 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. Putting them together we have a complete axiomatization of the constructor terms built from C.

The following proposition is a direct result of the axioms (Function) defined for every $c \in C$ in the specification MANYSORTED$\{\{\mathit{Term}\}, C, \emptyset\}$, which is imported by CONSTRUCTORS$\{C\}$.

Proposition 3.5. *Let M be any model such that $M \vDash$ CONSTRUCTORS$\{C\}$. Let $[\![\mathit{Term}]\!]_M = |[\![\mathit{Term}]\!]|$ be the carrier set of Term in M (see Notation 3.4). For any $c \in C$ with arity $n = \mathit{arity}(c)$, we define a function*

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

as follows:

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

Therefore, $f_c(a_1, \dots, a_n)$ is a singleton for every $a_1, \dots, a_n \in [\![\mathit{Term}]\!]_M$.

Remark 3.6. 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 _•_ given by M, we abuse the notation and also write $M_c(a_1, \dots, a_n)$ to mean $f_c(a_1, \dots, a_n)$, when M is given.

Proposition 3.7. *Let M be any model such that $M \vDash$ CONSTRUCTORS$\{C\}$. Let distinct $c, d \in C$, $n = \mathit{arity}(c)$, $m = \mathit{arity}(d)$. We define functions (see Remark 3.6):*

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

[2] And thus we answer a question 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 "no junk, no confusion" can be internalized?
[3] We only consider one-sorted case in this paper. For the general case, we refer the reader to the technical report [8].

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

Then by the axioms (No Confusion)*, we have that M_c, M_d are injective functions, and their ranges are disjoint.*

Proposition 3.8. *Let* $\mathbb{T}\mathrm{erm}$ *be the set of terms built from constructors in C. Then by the axiom* (Inductive Domain)*, $[\![Term]\!]_M$ is isomorphic to* $\mathbb{T}\mathrm{erm}$*, for any model $M \vDash \mathsf{CONSTRUCTORS}\{C\}$.*

3.3.1 Example: Natural Numbers

Natural numbers can be regarded as constructor terms of sort Nat built from two functions: $\mathbb{0} \colon \epsilon \to Nat$ and $\mathbb{s} \colon Nat \to Nat$. In the following, we show the specification of natural numbers, obtained by instantiating the constructor specification in Section 3.3 with $C = \{\mathbb{0}, \mathbb{s}\}$ and $Term = Nat$. We also inline all the imported MANYSORTED specification for better readability.

spec NAT
  Import: SORTS
  Axiom:
    (Sort Name) $(Nat \in [\![Sorts]\!]) \wedge (\exists z.\, Nat = z)$
    (Nonempty Inhabitant) $[\![Nat]\!] \neq \bot$
    (Function) $\exists y{:}Nat.\, \mathbb{0} = y$
    (Function) $\forall x{:}Nat.\, \exists y{:}Nat.\, \mathbb{s}\, x = y$
    (Function) $\forall x_1{:}Nat.\, \forall x_2{:}Nat.\, \exists y{:}Nat.\, plus\, x_1\, x_2 = y$
    (No Confusion) $\forall x{:}Nat.\, \mathbb{0} \neq \mathbb{s}\, x$
    (No Confusion) $\forall x{:}Nat.\, \forall y{:}Nat.\, \mathbb{s}\, x = \mathbb{s}\, y \to x = y$
    (Inductive Domain) $[\![Nat]\!] = \mu N.\, \mathbb{0} \vee \mathbb{s}\, N$
endspec

The above definition is self-explanatory. Note that the (Function) axioms specify that $\mathbb{0}$, $\mathbb{s}$, and *plus* are FOL-like functions. The (No Confusion) axioms specify that $\mathbb{0}$ is not the successor of any natural numbers and the successors of two numbers are equal only if the two numbers are equal. Finally, (Inductive Domain) states that the inhabitant set of Nat is the smallest set closed under $\mathbb{0}$ and $\mathbb{s}$.

3.4 A Discussion about Curried and Uncurried Styles

There are two ways to define multary functions and predicates in ML. In the above, we showed the curried style, where a multary function or predicate takes the arguments one by one. For example, the addition of x and y is represented by $plus\, x\, y$, where the function *plus* is applied to x first, yielding a partial evaluation result $plus\, x$. Then, the result is further applied to y, yielding the final result. An alternative is the uncurried style, where we define the addition of x and y as $plus\, (pair\, x\, y)$, where *pair* is the pairing symbol that builds the pair of x and y.

The curried style uses the built-in ML application construct so the encoding is simpler while the uncurried style requires some infrastructure definition such as *pair*. On the other hand, the uncurried style does not produce partial evaluation results such as $plus\, x$. All function and predicate applications must be complete. Therefore, the ML models of an uncurried style definition are often simpler to build than those of a curried style, because they do not need have elements that correspond to the semantics of those partial evaluation results. The uncurried style is also more preferable in a setting where we overload functions and predicates, because we can know the sorts of all arguments upfront.

In this paper, we focus on the curried style, but the reader can refer to [8] for several uncurried style definitions.

4 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 (with loose semantics) 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

4.1 Three Matching Logic Models of the Specification BNAT

We present four 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, the third one is similar to the first one but with a less conventional interpretation for $\mathbb{s}$ and le, and the fourth one is based on the set-theoretic definition of natural numbers.

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

model $\mathbb{M}1$ **of** BNAT
  Carrier Set M includes:
    $\mathbb{def}, \mathbb{inh}, \mathbb{N}\mathrm{at}, \mathbb{s}, \mathbb{le}$
    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:
    $\mathbb{M}1_{\lceil\_\rceil} = \{\mathbb{def}\}$ $\mathbb{M}1_{[\![\_]\!]} = \{\mathbb{inh}\}$ $\mathbb{M}1_{\mathit{Sorts}} = \{\mathbb{N}\mathrm{at}\}$
    $\mathbb{M}1_{\mathit{Nat}} = \{\mathbb{N}\mathrm{at}\}$ $\mathbb{M}1_{\mathbb{0}} = \{0\}$ $\mathbb{M}1_{\mathbb{s}} = \{\mathbb{s}\}$ $\mathbb{M}1_{\mathit{le}} = \{\mathbb{le}\}$
  Application Interpretation:
    $\mathbb{def} \bullet a = M$, for all $a \in M$
    $\mathbb{inh} \bullet \mathbb{N}\mathrm{at} = \mathbb{N}$
    $\mathbb{s} \bullet n = \{n+1\}$, 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:

$\mathbb{def}, \mathbb{inh}, \mathbb{N}\mathrm{at}, \mathbb{s}, \mathbb{le}$

n, for $n \in \mathbb{N}$ where $\mathbb{N}$ is the set of natural numbers

∞, a distinguished infinity symbol

$\mathbb{le} \rightsquigarrow n$, for $n \in \mathbb{N}$

$\mathbb{le} \rightsquigarrow \infty$

Symbol Interpretation:

$\mathbb{M}2_{\lceil\_\rceil} = \{\mathbb{def}\}$ $\quad \mathbb{M}2_{[\![\_]\!]} = \{\mathbb{inh}\}$ $\quad \mathbb{M}2_{Sorts} = \{\mathbb{N}\mathrm{at}\}$

$\mathbb{M}2_{Nat} = \{\mathbb{N}\mathrm{at}\}$ $\quad \mathbb{M}2_{\mathbb{0}} = \{0\}$ $\quad \mathbb{M}2_{\mathbb{s}} = \{\mathbb{s}\}$ $\quad \mathbb{M}2_{le} = \{\mathbb{le}\}$

Application Interpretation:

$\mathbb{def} \bullet a = M$, for all $a \in M$

$\mathbb{inh} \bullet \mathbb{N}\mathrm{at} = \mathbb{N} \cup \{\infty\}$

$\mathbb{s} \bullet n = \{n+1\}$, 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:

$\mathbb{def}, \mathbb{inh}, \mathbb{N}\mathrm{at}, \mathbb{s}, \mathbb{le}$

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:

$\mathbb{M}3_{\lceil\_\rceil} = \{\mathbb{def}\}$ $\quad \mathbb{M}3_{[\![\_]\!]} = \{\mathbb{inh}\}$ $\quad \mathbb{M}3_{Sorts} = \{\mathbb{N}\mathrm{at}\}$

$\mathbb{M}3_{Nat} = \{\mathbb{N}\mathrm{at}\}$ $\quad \mathbb{M}3_{\mathbb{0}} = \{0\}$ $\quad \mathbb{M}3_{\mathbb{s}} = \{\mathbb{s}\}$ $\quad \mathbb{M}3_{le} = \{\mathbb{le}\}$

Application Interpretation:

$\mathbb{def} \bullet a = M$, for all $a \in M$

$\mathbb{inh} \bullet \mathbb{N}\mathrm{at} = \mathbb{R}_{\geq 0}$

$\mathbb{s} \bullet r = \{r+1\}$, 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

The Fourth Matching Logic Model $\mathbb{M}4$ of BNAT This is a more convoluted model based on the set-theoretic definition of natural numbers. In this definition, the natural numbers are defined in the following way (also known as the von Neumann ordinals):

$$\begin{aligned}
\overline{0} &\equiv \{\}, \\
\overline{1} &\equiv \{\overline{0}\}, \\
\overline{2} &\equiv \{\overline{0}, \overline{1}\}, \\
\overline{3} &\equiv \{\overline{0}, \overline{1}, \overline{2}\}, \ldots
\end{aligned}$$

where $\{\}$ is the empty set. Let us define $\overline{\mathbb{N}} = \{\overline{0}, \overline{1}, \dots\}$ be the set of set-theoretic natural numbers. Note that under this definition, the less-than relation between natural numbers becomes set inclusion, i.e., $n_1 \leq n_2$ iff $\overline{n_1} \subseteq \overline{n_2}$.

Next, we define the ML model $\mathbb{M}4$.

model $\mathbb{M}4$ **of** BNAT
  Carrier Set M includes:
    $\mathbb{def}, \mathbb{inh}, \mathbb{N}\mathrm{at}, \mathbb{s}, \mathbb{le}$
    $\overline{n}$, for $\overline{n} \in \overline{\mathbb{N}}$
    $\mathbb{le} \rightsquigarrow \overline{n}$, for $\overline{n} \in \overline{\mathbb{N}}$
  Symbol Interpretation:
    $\mathbb{M}4_{\lceil \_ \rceil} = \{\mathbb{def}\}$ $\mathbb{M}4_{\llbracket \_ \rrbracket} = \{\mathbb{inh}\}$ $\mathbb{M}4_{Sorts} = \{\mathbb{N}\mathrm{at}\}$
    $\mathbb{M}4_{Nat} = \{\mathbb{N}\mathrm{at}\}$ $\mathbb{M}4_{\mathbb{0}} = \{\overline{0}\}$ $\mathbb{M}4_{\mathbb{s}} = \{\mathbb{s}\}$ $\mathbb{M}4_{le} = \{\mathbb{le}\}$
  Application Interpretation:
    $\mathbb{def} \bullet a = M$, for all $a \in M$
    $\mathbb{inh} \bullet \mathbb{N}\mathrm{at} = \overline{\mathbb{N}}$
    $\mathbb{s} \bullet \overline{n} = \{\overline{n+1}\}$, for all $\overline{n} \in \overline{\mathbb{N}}$
    $\mathbb{le} \bullet \overline{n} = \{\mathbb{le} \rightsquigarrow \overline{n}\}$, for all $\overline{n} \in \overline{\mathbb{N}}$
    $(\mathbb{le} \rightsquigarrow \overline{n_1}) \bullet \overline{n_2} = M$, if $\overline{n_1} \subseteq \overline{n_2}$ for $n_1, n_2 \in \overline{\mathbb{N}}$
    $a \bullet b = \emptyset$, if none of the above applies, for $a, b \in M$
endmodel

4.2 Explaining the Interpretation of Patterns

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. We do not consider the model $\mathbb{M}4$ here because its representations of natural numbers are based on set-theoretic encodings and is too different to be compared with the other three. Recall that $\neg_{Nat}(\mathbb{s}\,\mathbb{0}) \equiv \llbracket Nat \rrbracket \wedge \neg(\mathbb{s}\,\mathbb{0})$ 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:

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

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 a 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:

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

Recall that $\llbracket Nat \rrbracket_{\mathbb{M}}$ is the inhabitant of Nat in $\mathbb{M}$, defined in Notation 3.4.

Next, we consider the case where $\rho(x) \in \llbracket Nat \rrbracket_{\mathbb{M}}$, 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:

$$|x \wedge le\,(\mathbb{s}\,\mathbb{0})\,x|_{\mathbb{M}1,\rho_0} = \{\rho_0(x)\} \cap |le\,(\mathbb{s}\,\mathbb{0})\,x|_{\mathbb{M}1,\rho_0} = \{0\} \cap \emptyset = \emptyset,$$

$$
\begin{aligned}
&|x \wedge le\,(\mathtt{s}\,\mathbb{0})\,x|_{\mathbb{M}2,\rho_0} = |x \wedge le\,(\mathtt{s}\,\mathbb{0})\,x|_{\mathbb{M}3,\rho_0} = \emptyset, \text{for the same reason as above,}\\
&|x \wedge le\,(\mathtt{s}\,\mathbb{0})\,x|_{\mathbb{M}1,\rho_3} = \{\rho_3(x)\} \cap |le\,(\mathtt{s}\,\mathbb{0})\,x|_{\mathbb{M}1,\rho_3} = \{3\} \cap M = \{3\},\\
&|x \wedge le\,(\mathtt{s}\,\mathbb{0})\,x|_{\mathbb{M}2,\rho_3} = |x \wedge le\,(\mathtt{s}\,\mathbb{0})\,x|_{\mathbb{M}3,\rho_3} = \{3\}, \text{for the same reason as above.}
\end{aligned}
$$

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 $[\![Nat]\!]_{\mathbb{M}1} = \mathbb{N}$:

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

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 last set above is not a countable set.

5 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 5.1 ([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:

$$
\begin{array}{ll}
(\textsc{Post-Fixpoint}) & \vdash \nu X.\, \varphi \to \varphi[\nu X.\, \varphi/X]\\[2ex]
(\textsc{Knaster-Tarski}) & \dfrac{\vdash \psi \to \varphi[\psi/X]}{\vdash \psi \to \nu X.\, \varphi}
\end{array}
$$

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$_\bot$)	$C[\bot] \to \bot$
	(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).

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 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 monotone function $\mathcal{F}$ over a complete lattice. For instance, the functional $\mathcal{F}$ for the natural numbers is given by $\mathcal{F}(X) = \{\mathbb{0}\} \cup \{\mathbb{s}\, x \mid x \in X\}$, defined over the powerset lattice. The set of natural numbers, defined in this way, is $\mu\mathcal{F} = \{\mathbb{0}, \mathbb{s}\,\mathbb{0}, \mathbb{s}^2\,\mathbb{0}, \ldots\}$. 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 $\{\mathbb{0}\} \cup \{\mathbb{s}(x) \mid x \in X_\phi\} \subseteq X_\phi$, i.e., we have to check $\phi(\mathbb{0})$ (base case) and that $\phi(x) \implies \phi(\mathbb{s}(\mathbb{0}))$ (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 $\varphi \equiv \mathbb{0} \vee \mathbb{s}\, X$ and $\varphi[\psi/X] \equiv \mathbb{0} \vee \mathbb{s}\,\psi$. It follows that $\varphi[\psi/X] \to \psi$ is equivalent to $\mathbb{0} \to \psi$ and $\mathbb{s}\,\psi \to \psi$, which can be informally expressed as $\psi(0)$ and $\psi(x) \to \psi(\mathbb{s}\, x)$. $\square$

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 following coinduction principle, which we will use in Sections 7 and 9, is dual to induction principle:

Complete Lattices	Matching Logic
$\dfrac{X \subseteq \mathcal{F}(X)}{X \subseteq \nu\mathcal{F}}$ (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 that $X_\phi \subseteq \nu\mathcal{F}$, 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 elements 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 the 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-fixpoint: $X \subseteq \mathcal{F}(X)$;
3. show that $X_\phi \subseteq X$.

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

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

□

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

7 Defining Types as Matching Logic Specifications

In this section, we show how to define types as matching logic specifications. We discuss the basic types such as Booleans and numbers in Section 7.1. Then, we discuss parameterized types in Section 7.2, fixed-length vector types in Section 7.3, parametric finite sets in Section 7.4, product dependent types in Section 7.5, sum dependent types in Section 7.6. We give some discussion on binding in Section 7.7.

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: $\mathit{Bool}, \mathit{tt}, \mathit{ff}, !, \&$
  Metavariable: patterns φ_1, φ_2
  Notation: $\varphi_1 \mathbin{\&} \varphi_2 \equiv \& \, \varphi_1 \, \varphi_2$
  Axiom:
    (Sort Name): $\mathit{Bool} \in [\![\mathit{Sorts}]\!]$
    (Function):
      $\mathit{ff} : \epsilon \to \mathit{Bool}$ $\mathit{tt} : \epsilon \to \mathit{Bool}$
      $! : \mathit{Bool} \to \mathit{Bool}$ $\& : \mathit{Bool} \times \mathit{Bool} \to \mathit{Bool}$
    (Inductive Domain): $[\![\mathit{Bool}]\!] = \mathit{tt} \vee \mathit{ff}$
    (No Confusion): $\mathit{tt} \neq \mathit{ff}$
    (Definition):
      $!\,\mathit{tt} = \mathit{ff}$ $!\,\mathit{ff} = \mathit{tt}$
      $\forall x{:}\mathit{Bool}.\, x \mathbin{\&} \mathit{tt} = x$ $\forall x{:}\mathit{Bool}.\, x \mathbin{\&} \mathit{ff} = \mathit{ff}$
      $\forall x{:}\mathit{Bool}.\, \mathit{tt} \mathbin{\&} x = x$ $\forall x{:}\mathit{Bool}.\, \mathit{ff} \mathbin{\&} x = \mathit{ff}$
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):
      $\mathbb{0}\colon \epsilon \to Nat$          $\mathbb{s}\colon Nat \to Nat$
    (INDUCTIVE DOMAIN): $[\![Nat]\!] = \mu X.\, \mathbb{0} \vee \mathbb{s}\, X$
    (NO CONFUSION):
      $\forall x{:}Nat.\, \mathbb{0} \neq \mathbb{s}\, x$
      $\forall x, y{:}Nat.\, \mathbb{s}\, x = \mathbb{s}\, y \to x = y$
endspec
```

Therefore, $[\![Nat]\!]$ is the smallest set built from $\mathbb{0}$ and $\mathbb{s}$, which are the only two constructors of Nat.

Proposition 7.1. *The following propositions hold:*

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

Explanation. We prove Item 1 as an example. 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 $\mathsf{NAT} \vDash \mathbb{0} \in [\![Nat]\!]$. □

Proposition 7.2. *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 any $n \in [\![Nat]\!]_M$, $M_{\mathbb{s}} \bullet n$ is a singleton, whose element we denote (by abuse of notation) as $\mathbb{s}(n)$. 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 7.3 (Successor Pre-fixpoint)**.** *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

$$|\mathbb{s}\, P \to P|_\rho = M$$

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

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

Proposition 7.4 (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 $n \in \mathbb{N}$, $\mathbb{s}(n) \in \rho(P)$. By Proposition 7.2, we have $\rho(P) = [\![Nat]\!]_M$, and thus $|\forall x{:}Nat.\, x \in P|_\rho = M$. □

Remark 7.5. 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:

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

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 (coproduct) 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:
    $s_1 \otimes s_2 \equiv \otimes\, s_1\, s_2$
    $\langle x, y\rangle \equiv \langle\_, \_\rangle\, x\, y$
  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. □

Proposition 7.6. *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 corresponding (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:
(Sum Sort)
$s_1 \in [\![Sorts]\!] \wedge s_2 \in [\![Sorts]\!] \to s_1{\oplus}s_2 \in [\![Sorts]\!]$
(Inject Left)
$\iota_1 : s_1 \to s_1{\oplus}s_2$
(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)
$\forall s_1, s_2{:}Sorts.\, [\![s_1{\oplus}s_2]\!] \subseteq (\iota_1\, [\![s_1]\!]) \vee (\iota_2\, [\![s_2]\!])$
endspec

Explanation. (Inject _) and (Eject _) are instances of (Function) and (Partial Function), respectively. □

Proposition 7.7. *The following hold:*

1. ι_1 *and* ι_2 *are injective functions.*
2. $[\![s_1{\oplus}s_2]\!] = (\iota_1\, [\![s_1]\!]) \vee (\iota_2\, [\![s_2]\!])$.
3. SUM$\{s_1, s_2\} \vdash \forall s_1, s_2{:}Sorts.\, (\iota_1\, [\![s_1]\!]) \wedge (\iota_2\, [\![s_2]\!]) = \bot$.

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 _).
3. Intuitively, assume $y \in (\iota_1\,[\![s_1]\!]) \wedge (\iota_2\,[\![s_2]\!])$ and we show contradiction. By assumption, there exist x_1 and x_2 such that $y = \iota_1(x_1) = \iota_2(x_2)$. We have $\epsilon_2(\iota_1(x_1)) = \bot = \epsilon_1(\iota_2(x_2))$ by (Inverse InjEj2). But $\iota_1(x_1) = y = \iota_2(x_2)$ and hence $\epsilon_2(\iota_1(x_1)) = \epsilon_2(\iota_2(x_2)) = x_2$ and $\epsilon_1(\iota_1(x_2)) = \epsilon_1(\iota_1(x_1)) = x_1$ by (Inverse InjEj1), which is a contradiction. □

Proposition 7.8. *$[\![s_1 \oplus s_2]\!] = [\![s_1]\!] \uplus [\![s_2]\!]$, 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 $s_1 \ominus\!\!\to s_2$, called the *function sort from s_1 to s_2*, as follows:

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

Proposition 7.9. *The following hold:*

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

Remark 7.10. Even if strongly related, there is a difference between $f{:}s_1 \ominus\!\!\to s_2$ and $f : s_1 \to s_2$. The former says that $f \in [\![s_1 \ominus\!\!\to s_2]\!]$ and the latter is a sugar syntax for the axiom

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

which is equivalent to

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

The relationship between the two notations is easy to see if we note that the definition of $[\![s_1 \ominus\!\!\to s_2]\!]$ can be written as $\exists f.\, f \wedge f\colon s_1 \to s_2$. However, $f \in [\![s_1 \ominus\!\!\to s_2]\!]$ says further that f is a functional symbol.

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

Proposition 7.11 (Natural Numbers Iteration Principle)**.**

$$\begin{aligned}&\forall h.\, \forall c{:}s.\, \forall f{:}s \ominus\!\!\!\rightarrow s.\, (h\, \mathbb{0} = c \wedge \forall n{:}Nat.\, h\,(\mathbb{s}\, n) = f\,(h\, n)) \rightarrow \\ &\qquad (\forall n{:}Nat.\, \exists\, y{:}s.\, h\, n = y)\end{aligned} \tag{ItNat}$$

Explanation. (ItNat) is equivalent to

$$\begin{aligned}&\forall h.\, \forall c{:}s.\, \forall f{:}s \ominus\!\!\!\rightarrow s.\, (h\, \mathbb{0} = c \wedge \forall n{:}Nat.\, h\,(\mathbb{s}\, n) = f\,(h\, n)) \rightarrow \\ &\qquad (\llbracket Nat \rrbracket \subseteq \exists x.\, \exists\, y{:}s.\, x \wedge h\, x = y)\end{aligned}$$

and we apply then the induction principle:

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

□

Example 7.12. The following ML specification defines two functions *plus* and *mult* on natural numbers in the usual way:

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

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

Proposition 7.13 (Natural Numbers (Primitive) Recursion Principle)**.**

$$\begin{aligned}&\forall h.\, \forall c{:}s.\, \forall g{:}(s \otimes Nat) \ominus\!\!\!\rightarrow s.\, (h\, \mathbb{0} = c \wedge \forall n{:}Nat.\, h\,(\mathbb{s}\, n) = g\,(h\, n)\, n) \rightarrow \\ &\qquad (\forall n{:}Nat.\, \exists\, y{:}s.\, h\, n = y)\end{aligned} \tag{PrRecNat}$$

Explanation. (PrRecNat) is equivalent to

$$\begin{aligned}&\forall h.\, \forall c{:}s.\, \forall g{:}(s \otimes Nat) \ominus\!\!\!\rightarrow s.\, (h\, \mathbb{0} = c \wedge \forall n{:}Nat.\, h\,(\mathbb{s}\, n) = g\,((h\, n)\, n)) \rightarrow \\ &\qquad (\llbracket Nat \rrbracket \subseteq \exists x.\, \exists\, y{:}s.\, x \wedge h\, x = y)\end{aligned}$$

and we apply then the induction principle:

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

□

Example 7.14. The following ML specification defines the factorial function *fact* in the usual way:

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

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

7.2.4 Parameterized (Finite) Lists

The type of parametric lists is a canonical example of a polymorphic datatype, i.e., a datatype parameterized by another type. Polymorphic datatypes are included in many programming languages (Java, C++, Haskell, etc.), known also as generic types. For instance, they were introduced to C++ in 1987, without a serious consideration on the 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 parametric lists, which can be used as a foundation for any implementation.

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

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

A reader familiar with a functional programming language perhaps prefers 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, it is not sufficient for implementing the datatype.

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

spec LIST$\{s\}$
  Import: SORTS
  Symbol: $List, nil, cons$
  Metavariable: element variables $x{:}s, x'{:}s, \ell{:}List\langle s\rangle, \ell'{:}List\langle s\rangle$
  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):

$nil \neq cons\, x\, \ell$

$cons\, x\, \ell = cons\, x'\, \ell' \to x = x' \wedge \ell = \ell'$

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 constants $cons$ and nil to functional interpretations. 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 finitely many 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 7.15 (Lists Induction Principle)**.**

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

Explanation.

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

□

Remark 7.16. Using a notation similar to that from Remark 7.5, (IndList) can be rewritten in the more familiar form:

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

Proposition 7.17 (Lists Iteration Principle)**.**

$$\begin{aligned}\mathsf{LIST}\{s\} \models &\forall h.\, \forall c{:}s'.\, \forall f{:}s \otimes s' \mathrel{\bigcirc\mkern-15mu\rightarrow} s'.\\ &\quad (h\, nil = c \wedge \forall x{:}s.\, \forall \ell{:}List\langle s\rangle.\, h\,(cons\, x\, y) = f\,(x, h\,\ell)) \to\\ &\quad (\forall \ell{:}List\langle s\rangle.\, \exists\, y{:}s'.\, h\,\ell = y)\end{aligned} \qquad \text{(ItList)}$$

Explanation. (ItList) is equivalent to

$$\begin{aligned}\mathsf{LIST}\{s\} \models &\forall h.\, \forall c{:}s'.\, \forall f{:}s \otimes s' \mathrel{\bigcirc\mkern-15mu\rightarrow} s'.\\ &\quad (h\, nil = c \wedge \forall x{:}s.\, \forall \ell{:}List\langle s\rangle.\, h\,(cons\, x\, y) = f\,(x, h\,\ell)) \to\\ &\quad ([\![List\langle s\rangle]\!] \subseteq \exists x.\, \exists\, y{:}s'.\, x \wedge h\, x = y)\end{aligned}$$

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

$$\dfrac{\dfrac{c{:}s' \qquad h\,nil = c}{\mathsf{LIST}\{s\} \models nil \in \begin{array}{c}\exists x.\,\exists\,y{:}s.\,x\\ \wedge\\ h\,x = y\end{array}}\textsc{Hyp} \qquad \dfrac{\dfrac{\dfrac{f}{\mathsf{LIST}\{s\} \models \begin{array}{l}\exists x.\,\exists\\ \to\\ \exists x.\,\exists\end{array}}}{\mathsf{LIST}\{s\} \models \begin{array}{l}\exists x.\,\exists\,y\\ \to\\ \exists x.\,\exists\,y\end{array}}}{\mathsf{LIST}\{s\} \models \begin{array}{l}\exists x.\,\exists\\ \to\\ cons\end{array}}}{\mathsf{LIST}\{s\} \models [\![List\langle s\rangle]\!] \subseteq \exists x.\,\exists\,y{:}s'.\,x}$$

$\square$

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

Example 7.18. Let $\mathsf{MAP}\{s\}$ be the following ML specification:

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

Then we obtain

$$\mathsf{MAP}\{s\} \models map \in [\![((s\ominus s') \otimes List\langle s\rangle)\ominus List\langle s'\rangle]\!]$$
$$\mathsf{MAP}\{s\} \models map\; g \in [\![List\langle s\rangle\ominus List\langle s'\rangle]\!]$$

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

Proposition 7.19 (Lists (Primitive) Recursion Principle)**.**

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

Explanation. We write (PrRecList) in the equivalent form

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

and apply the induction principle for lists:

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

□

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

Example 7.20. Let $\mathsf{FOLDR}\{s\}$ be the following ML specification:

spec $\mathsf{FOLDR}\{s\}$
  Import: $\mathsf{LIST}\{s\}$
  Symbol: $foldr$
  Metavariable: element variables $x{:}s, z{:}s', \ell{:}List\langle s\rangle, f{:}s \otimes s' \ominus\!\!\!\!\rightarrow s'$
  Axiom:
    $foldr\,f\,z\,nil = z$
    $foldr\,f\,z\,(cons\,x\,\ell) = f\,x\,(foldr\,f\,z\,\ell)$
endspec

Then we obtain

$$\mathsf{FOLDR}\{s\} \models foldr{:}((s \ominus\!\!\!\!\rightarrow s') \otimes s' \otimes List\langle s\rangle) \ominus\!\!\!\!\rightarrow s'$$
$$\mathsf{FOLDR}\{s\} \models foldr\,f\,z{:}List\langle s\rangle \ominus\!\!\!\!\rightarrow s'$$

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

7.2.5 Parameterized (Infinite) Streams

The type of streams (infinite lists) is a canonical example of coinductive types with associated 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

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

or a Haskell-like notation:

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

where the constructor $x ::: \ell$ corresponds to $cons(x, \ell)$. 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 $hd(cons(x, \ell)) = x$ and $tl(cons(x, \ell)) = \ell$. Following the convention of inductive and coinductive data types, we use *constructors* for (inductive) data and *destructors* for codata. In some literature, destructors are also called *observers*.

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 while the destructors are defined axiomatically.

spec STREAM$\{s\}$
  Symbol: $\mathit{Stream}, \mathit{cons}, \mathit{hd}, \mathit{tl}, \approx_{\mathit{Stream}}$
  Metavariable: element variables $x, x'{:}s; \ell, \ell_1, \ell_2{:}\mathit{Stream}\langle s\rangle$
  Notation:
    $\mathit{Stream}\langle s\rangle \equiv \mathit{Stream}\ s$
    $\ell_1 \approx_{\mathit{Stream}} \ell_2 \equiv \langle \ell_1, \ell_2\rangle \in \approx_{\mathit{Stream}}$
    $\mathit{cons}\, x\, \langle \ell_1, \ell_2\rangle \equiv \langle \mathit{cons}\, x\, \ell_1, \mathit{cons}\, x\, \ell_2\rangle$
    $\alpha(X) \equiv \exists \ell.\, \ell \wedge \mathit{hd}\, \ell \in [\![s]\!] \wedge \mathit{tl}\, \ell \in X$
    $\beta(R) \equiv \exists \ell, \ell'{:}\mathit{Stream}\langle s\rangle.\, \langle \ell, \ell'\rangle \wedge \mathit{hd}\, \ell = \mathit{hd}\, \ell' \wedge \langle \mathit{tl}\, \ell, \mathit{tl}\, \ell'\rangle \in R$
  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)
      $\mathit{cons}\, x\, \ell = \mathit{cons}\, x'\, \ell' \to x = x' \wedge \ell = \ell'$
    (Destructors)
      $\mathit{hd}(\mathit{cons}\, x\, \ell) = x$
      $\mathit{tl}(\mathit{cons}\, x\, \ell) = \ell$
    (Bisimilarity)
      $\approx_{\mathit{Stream}} = \nu R{:}\mathit{Stream}\langle s\rangle \otimes \mathit{Stream}\langle s\rangle.\, \mathit{cons}\, [\![s]\!]\, R$
      $\forall \ell_1, \ell_2{:}\mathit{Stream}\langle s\rangle.\, (\ell_1 \approx_{\mathit{Stream}} \ell_2) = (\ell_1 = \ell_2)$
endspec

Explanation. The axioms for sorts and constructors are similar to those for 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, $|\nu X.\, \mathit{cons}\, [\![s]\!]\, X|_\rho = \nu \mathcal{F}^\rho_{X,\varphi}$, where $\varphi \equiv \mathit{cons}\, [\![s]\!]\, X$. Since $\mathcal{F}^\rho_{X,\varphi}$ is cocontinuous, we have

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

where $A ::: B \equiv (|\mathit{cons}|_\rho \bullet A) \bullet B$ 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 an i such that $a_i \neq b_i$, by applying (No Confusion) $i+1$ times.. It is easy to see now the similarity with the Haskell definition of infinite trees. Note that the definition does not depend 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 $R{:}\mathit{Stream}\langle s\rangle \otimes \mathit{Stream}\langle s\rangle$, the greatest fixpoint can be computed starting from $[\![s]\!]^\infty_M \times [\![s]\!]^\infty_M$:

$$\begin{aligned}
\nu \mathcal{F}^\rho_{R,\varphi} &= [\![s]\!]^\infty_M \times [\![s]\!]^\infty_M \cap \mathcal{F}^\rho_{R,\varphi}([\![s]\!]^\infty_M \times [\![s]\!]^\infty_M) \cap \mathcal{F}^\rho_{R,\varphi}(\mathcal{F}^\rho_{R,\varphi}([\![s]\!]^\infty_M \times [\![s]\!]^\infty_M)) \cap \cdots \\
&= [\![s]\!]^\infty_M \times [\![s]\!]^\infty_M \cap |\mathit{cons}\, [\![s]\!]\, X|_{\rho[[\![s]\!]^\infty_M \times [\![s]\!]^\infty_M / R]} \cap \cdots \\
&= [\![s]\!]^\infty_M \times [\![s]\!]^\infty_M \cap [\![s]\!]_M ::: [\![s]\!]^\infty_M \times [\![s]\!]^\infty_M \cap [\![s]\!]_M ::: [\![s]\!]_M ::: [\![s]\!]^\infty_M \times [\![s]\!]^\infty_M \cap \cdots
\end{aligned}$$

$$= [\![s]\!]_M^\infty \times [\![s]\!]_M^\infty \cap \{\langle a_0 :: \ell, a_0 :: \ell'\rangle \mid a_0 \in [\![s]\!]_M, \langle \ell, \ell'\rangle \in [\![s]\!]_M^\infty \times [\![s]\!]_M^\infty\} \cap \cdots$$
$$= \{\langle a_0 ::: a_1 ::: a_2 ::: \cdots,\ a_0 ::: a_1 ::: a_2 ::: \cdots\rangle \mid a_i \in [\![s]\!]_M, i = 0, 1, 2, \ldots\}$$

where φ is now $\mathit{cons}\,[\![s]\!]\,R$. Note 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. □

Proposition 7.21. *The following results show that streams can be equivalently specified using destructors.*

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

Explanation. Item 1 shows that destructors and constructors are inverse to 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 alone cannot construct streams (there is no *nil*-like constructor). But it can reconstruct a stream from its components given by the destructors. The notation $\beta(R)$ says that R is a bisimulation (see, e.g., [11]) or a behavioral equivalence (see, e.g., [12]). Then Item 4 specifies that $\approx_{\mathit{Stream}}$ is the largest bisimulation (behavioral equivalence). □

Proposition 7.22 (Streams Coinduction Principle I)**.**

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

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

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

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

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

Corollary 7.23 (Streams 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 Proposition 7.21. □

Proposition 7.24 (Streams Coiteration Principle)**.**

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

Explanation. Note the use of the existential quantifier, comparing with the dual universal quantifier used in Proposition 7.17. 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

$$\exists h.\, \exists x{:}s'.\, \exists c{:}s' \ominus\!\!\rightarrow s.\, \exists g{:}s' \ominus\!\!\rightarrow s'.\, P'.$$

We have:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{c \in [\![s' \ominus\!\!\rightarrow s]\!]}{c\,y \in [\![s]\!]}\,\textsc{Fol} \qquad \dfrac{g \in [\![s' \ominus\!\!\rightarrow s']\!]}{h\,(g\,y) \in P}\,\textsc{Fol}}{P' \to \alpha(P)}\,\textsc{Fol}}{P \to \alpha(P)}\,\exists\text{-}\textsc{Generalization}}{P \to \nu X.\, \alpha(X)}\,\textsc{Knaster-Tarski}}{P \to [\![\mathit{Stream}\langle s\rangle]\!]}\,\textsc{Proposition 7.21 Item 2}$$

□

Example 7.25. Given the following ML specification:

```
spec CNST&FROM
   Import: NAT + STREAM{Nat}
   Symbol: cnst, from
   Metavariable: element variables n:Nat
   Axiom:
      hd (cnst n) = n              hd (from n) = n
      tl (cnst n) = cnst n         tl (from n) = from (𝕤 n)
endspec
```

we obtain

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

by applying CoitStream. For instance, for *from* we take $c\,n = n$ and $g\,n = \mathbb{s}\,n$. The symbol *cnst* is used for specifying the constant stream given by the argument n, $\langle n, n, n, \ldots\rangle$, and the symbol *from* for specifying the stream of natural numbers starting from the argument n, $\langle n, n+1, n+2, \ldots\rangle$.

7.3 Fixed-Length Vector Types

A vector type (sort) $Vec\, s\, n$ 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: Vec, null
   Metavariable: element variables s:Sorts, n:Nat
   Axiom:
      (Sort Name): ∀n:Nat.∀s:Sorts. Vec s n ∈ ⟦Sorts⟧
      (Function):
         ∃y. Vec = y
         ∃y. null = y
      (Inductive Domain):
         ⟦Vec s 𝟘⟧ = null
         ⟦Vec s (𝕤 n)⟧ = ⟦s ⊗ Vec s n⟧
endspec
```

Explanation. (Sort Name) and (Function) are similar to the specification 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 $Vec\, s\, (\mathbb{s}\, n)$ is an alias for the product type of s and the vector type $Vec\, s\, n$. 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: $Vec, null, cons$
Metavariable: element variables $s{:}Sorts,\ n{:}Nat$
Axiom:
(Sort Name) : $\forall n{:}Nat. \forall s{:}Sorts.\ Vec\, s\, n \in [\![Sorts]\!]$
(Function) :
$\exists y.\ Vec = y$
$\exists y.\ null = y$
(Inductive Domain) :
$[\![\, Vec\, s\, \mathbb{0}]\!] = null$
$[\![\, Vec\, s\, (\mathbb{s}\, n)]\!] = cons\, [\![s]\!]\, [\![\, Vec\, s\, n]\!]$
(No Confusion)
$\forall x{:}s. \forall y{:}Vec\ \ s\, n.\ null \neq cons\, x\, y$
$\forall x, x'{:}s..\forall y, y'{:}Vec\, s\, n.\ cons\, x\, y = cons\, x'\, y' \to x = x' \wedge y = y'$
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 $Vec\, s\, (\mathbb{s}\, n)$ contains all the finite lists of length $\mathbb{s}\, n$ where the base sort is s. □

7.4 Parametric (Finite) Sets

Similarly to the type of parametric lists $List\langle s\rangle$, the type of parametric finite sets $Set\langle s\rangle$ is a parametric type that is parametric in a sort s. The following ML specification defines finite sets as terms built from *empty*, *union*, and *singleton* (that builds singleton sets from elements), where *empty* is the unit element of *union* and *union* satisfies the properties of associativity, commutativity, and idempotency. We also define the operation *delete* that deletes an element from a set.

spec SET$\{s\}$
imports : SORTS
Symbol: $Set, empty, union, singleton, delete$
Metavariable: element variables $s{:}Sorts$
Notation: $Set\langle s\rangle \equiv Set\, s$
Axiom:
(Sort Name) : $\forall s{:}Sorts.\ Set\langle s\rangle \in [\![Sorts]\!]$
(Function) :
$\exists y.\ Set = y$
$\exists y{:}Set\langle s\rangle.\ empty = y$
$\forall y_1, y_2{:}Set\langle s\rangle. \exists y{:}Set\langle s\rangle.\ union\, y_1\, y_2 = y$
$\forall x{:}s. \exists y{:}Set\langle s\rangle.\ singleton\, x = y$
$\forall x{:}s. \forall y{:}Set\langle s\rangle. \exists y'{:}Set\langle s\rangle.\ delete\, y\, x = y'$
(Inductive Domain) :
$[\![Set\langle s\rangle]\!] = \mu S.\ empty \vee (singleton\, [\![s]\!]) \vee union\, S\, S$
(Associativity) :

$(union\, y_1\, (union\, y_2\, y_3)) = ((union\, (union\, y_1\, y_2)\, y_3))$

(Commutativity) :

$(union\, y_1\, y_2) = (union\, y_2\, y_1)$

(Idempotency) :

$(union\, y\, y) = y$

(Unit) :

$(union\, y\, empty) = y$

(Delete) :

$delete\, empty\, z = empty$

$delete\, (singleton\, x)\, z = empty \wedge x = z \vee (singleton\, x) \wedge x \neq z$

$delete\, (union\, y_1\, y_2)\, z = union\, (delete\, y_1\, z)\, (delete\, y_2\, z)$

(No Confusion)

$\forall x{:}s.\ empty \neq singleton\, x$

$\forall x, y{:}s.\ singleton\, x = singleton\, y \to x = y$

$empty = (union\, y_1\, y_2) \to y_1 = empty \wedge y_2 = empty$

$\forall x{:}s.\ (singleton\, x) = (union\, y_1\, y_2)$
$\quad \to (y_1 = singleton\, x) \wedge (y_2 = singleton\, x)$
$\quad \vee (y_1 = singleton\, x) \wedge (y_2 = empty)$
$\quad \vee (y_1 = empty) \wedge (y_2 = singleton\, x)$

$(union\, y_1\, y_2) = (union\, y_3\, y_4)$
$\quad \to \forall z{:}s.\ (union\, (delete\, y_1\, z)\, (delete\, y_2\, z))$
$\qquad = (union\, (delete\, y_3\, z)\, (delete\, y_4\, z))$

endspec

Explanation. The axioms (Sort Name), (Function), and (Inductive Domain) are similar to those in the specification LIST$\{s\}$. The axioms (Associativity), (Commutativity), and (Idempotency) define the corresponding properties of the operation *union*. The axiom (Unit) defines that *empty* is the unit element of *union*. The axiom (Delete) defines the behaviors of the operation *delete*. The (No Confusion) axioms define the no-confusion properties modulo the above axioms. Note that the last (No Confusion) uses *delete* to reduce the sizes of the argument sets. □

In general, it is highly nontrivial to define the (No Confusion) axioms for constructors that have underlying axioms (such as the associativity and commutativity axioms for *union* above). In the above specification SET$\{s\}$, we define (No Confusion) using the operation *delete* that is not a constructor of sets, so the resulting specification is easier to understand, but it is specific to defining sets and cannot be generalized to define arbitrary constructors modulo any axioms.

There are two ways to define constructors modulo axioms in a more general way. Firstly, we can use *unification* and the most general unifiers (mgu) when they exist. Intuitively, unification solves equations between constructor terms and mgu is a complete and minimal solution (i.e., a substitution mapping variables to terms). Indeed, a (No Confusion) axiom has the following common form $c\, x_1\, \ldots\, x_n = d\, y_1\, \ldots\, y_m \to \varphi$, where φ specifies when the two terms (built from c and d respectively) are equal. If there are no underlying axioms about constructors, then it is clear that φ requires $c = d$, $m = n$, and all the corresponding arguments are equal. However, when there are underlying axioms about constructors, φ may be different. For example, the fourth (No Confusion) axiom in the specification SET$\{s\}$ states three conditions where $singleton\, x$ equals $union\, y_1\, y_2$, where *singleton* and *union* are different constructors of sets. A key observation is that φ can be generated from the mgu of $c\, x_1\, \ldots\, x_n$ and $d\, y_1\, \ldots\, y_m$. Thus, we can use unification and mgu to define the (No-Confusion) axioms. We leave further research on this approach as future work.

The drawback of the above approach using unification is that mgu may not exist for all underlying constructor axioms. Even if the mgu exists, the resulting (No Confusion) axioms may be hard to read and understood by humans. The second approach solves these problems by not defining the no-confusion properties modulo axioms directly on the constructor terms. Instead, we define the standard (No-Confusion) axioms assuming that there are no underlying axioms and then capture (No-Confusion) by defining the smallest *congruence relation* that includes all the (equational) properties about constructors. Then, for any two terms t_1 and t_2, t_1 equals to t_2 modulo the underlying axioms if and only if they are in the defined congruence relation. Following this approach, we essentially capture *initial algebra semantics* in an axiomatically way in ML. This topic is studied thoroughly in the technical report [8].

7.5 Dependent Product Types

The 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 $s_1 \ominus\!\!\!\!\to s_2$. However, it is (surprisingly) not a trivial task to capture the *binding behavior* of $\Pi x{:}s_1 .\, s_2$, in which x is bound in s_2. We shall leave this to Section 7.7. In this paper we only show how to specify the inhabitant set of $\Pi x{:}s_1 .\, s_2$. In other words, we only define $[\![\Pi x{:}s_1 .\, s_2]\!]$ (directly as a syntactic sugar) without defining $\Pi x{:}s_1 .\, s_2$, because the latter requires us to deal with the binding of x into 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 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 $s_1 \ominus\!\!\!\!\to s_2$. □

Proposition 7.26. *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. $\forall f{:}(\Pi x{:}s_1 .\, s_2(x)) .\, x \in [\![s_1]\!] \to f\,x \in [\![s_2(x)]\!]$.

7.6 Dependent Sum Types

The 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 sum type $\Sigma x{:}s_1 .\, s_2$ and a has type s_1, then b has type $s_2[a/x]$. 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 $s_1 \otimes s_2$.

Similarly to the dependent product 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):
$\forall x, x'{:}s_1 .\, \forall y{:}s_2(x) .\, \forall y'{:}s_2(x') .\, \langle x, y\rangle = \langle x', y'\rangle \to x = x' \wedge y = y'$
endspec

Proposition 7.27. *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* $\models \forall x{:}s_1 .\, \forall y{:}s_2(x) .\, \langle x, y\rangle \in [\![\Sigma x{:}s_1 .\, s_2(x)]\!]$.

7.7 Discussion: Defining Binders

In Sections 7.5 and 7.6, we showed how to define the inhabitant sets of dependent product and sum types in ML. However, we did not show the complete ML definition of any type systems because type systems have terms with binders. In [3], we proposed a general approach to defining arbitrary binders in ML and we developed sound and complete ML specifications for various logical systems with binders, including λ-calculus, System F, pure type systems, and π-calculus. In this section, we only discuss the high-level ideas about how to handle binders in ML.

It is known that binders are difficult to be encoded in a formal system. The difficulty lies in the handling of α-equivalence and capture-avoiding substitution. Let us use (untyped) λ-calculus as an example. In λ-calculus, function abstraction $\lambda x.\, e$ is a binding construct that binds x in e. As a result, one function abstraction can have many syntactically different representations. For example, the identity function can be represented as $\lambda x.\, x$, or $\lambda y.\, y$, or $\lambda z.\, z$, etc. Renaming bound variables in a function abstraction does not change the meaning of the function. Therefore, we are interested in the syntax of λ-calculus *modulo* the renaming of bound variables, i.e., α-equivalence. Let us look at the following axiom (schema) in λ-calculus, called the (Beta) axiom that defines the behavior of function application:

$$(\textsc{Beta}) \quad (\lambda x.\, e)\, e' = e[e'/x]$$

where x is a variable, e and e' are λ-calculus expressions, and $e[e'/x]$ denotes the result of substituting e' for x in e, where bound variables are automatically renamed to prevent *variable capture*. Variable capture means that a free variable in e' gets bound by a binder in e after substitution. For example, consider the function application: $(\lambda x.\, \lambda y.\, x)\, y$. If we apply (Beta) without the necessary variable renaming, we will get the incorrect result $\lambda y.\, y$, where the argument y is now bound by λy. The correct way requires us to rename $\lambda x.\, \lambda y.\, x$ to $\lambda x.\, \lambda z.\, x$, where z is a fresh variable that does not occur in the function nor in the argument. Then, we have $(\lambda x.\, \lambda z.\, x)\, y$, which then yields the correct result $\lambda z.\, y$. This is called the capture-avoiding substitution.

To define $\lambda x.\, e$ in ML, we first define the set $\{\langle y, e[y/x]\rangle \mid \text{for all variables } y\}$ that includes all possible α-equivalent representations of the function $\lambda x.\, e$. Note that the above set is essentially the *graph* of the function $x \mapsto e$. Therefore, we use the following ML pattern to define the graph set:

$$\langle x, e\rangle \equiv \mathit{pair}\, x\, e$$
$$[x]e \equiv \mathsf{intension}\, \exists x.\, \langle x, e\rangle$$

where *pair* is the pairing symbol that takes x and e and returns the pair $\langle x, e\rangle$. Pattern $\exists x.\, \langle x, e\rangle$ then ranges over all variables x and computes the union set, which is exactly the graph set. Also note that in $\exists x.\, \langle x, e\rangle$, the variable x is bound in e by the ML's built-in binder $\exists$. Thus, we are using the built-in $\exists$ binder to capture the binding behavior of λ in λ-calculus. The construct $\mathsf{intension}$ takes a set (as a collection of elements) and returns the set itself as one individual element. Therefore, the pattern $\mathsf{intension}\, \exists x.\, \langle x, e\rangle$, or using the notation $[x]e$, represents the graph set as one individual element instead of a collection of the argument-value pairs, and thus when we apply other functions or constructors to the graph $[x]e$, we will not trigger ML's pointwise extension (see Definition 2.4). For more technical details, we refer the reader to [3, Section 6].

In short, we can use the ML pattern $[x]e$ to denote the graph of the function $x \mapsto e$. The graph pattern includes all the binding information of x into e. Note that the notation $[x]e$ is a common notation in other approaches to dealing with binders using higher-order abstract syntax [**?**] and nominal logic [**?**]. Finally, we can define $\lambda x.\, e \equiv \mathsf{lambda}\, [x]e$ as syntactic sugar, where we apply the constructor lambda to the graph set $[x]e$. Different binders will have different outmost constructors, but all take the same graph set as argument (if they have the same binding behavior of x into e).

Now, let LAMBDA be the ML specification that includes the above binder definitions plus the (Beta) axiom. Then we can prove that LAMBDA precisely captures λ-calculus. Formally, we prove the following conservative extension theorem (see [3, Theorem 36]):

$$\vdash_\lambda e_1 = e_2 \quad \text{iff} \quad \mathsf{LAMBDA} \vdash e_1 = e_2$$

where $\vdash_\lambda$ denotes the formal reasoning of λ-calculus, i.e., equational reasoning modulo axiom (Beta).

8 Defining Basic Process Algebra as Matching Logic Specifications

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 the 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 processes can be specified using guarded recursive specifications. We strongly believe that the approach used for BPA can be successfully applied to the other theories and ML is a suitable framework for describing and reasoning about processes.

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:

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

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

In the following, we define the ML specification for BPA.

spec BPA

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

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

Notation:

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

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

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

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

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

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) :

$x + y = x' + y' \to x = x' \land y = y'$

$x\,; y = x'\,; y' \to x = x' \land y = y'$

(Transition) :

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

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

$\bullet_u y = \mu Y.\, (\bullet_u \surd)\,; y \lor Y + [\![PTerm]\!] \lor [\![PTerm]\!] + Y \lor$
$\qquad\qquad \exists x, x', y'{:}PTerm.\, x\,; y' \land y = x'\,; y' \land x \in \bullet_u x'$

(Bisimulation)

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

endspec

Explanation. The first (Transition) axiom is equivalent to $\bullet[\![Atom]\!]\,[\![PTerm]\!]) \subseteq [\![PTerm]\!] \lor \surd$ and specifies the signature of the transition relation. The second (Transition) axiom is the definition of the predicate $\surd$. The third (Transition) axiom says that $\bullet_u y$ denotes the u-predecessors w.r.t. the transition relation of a process term y; the correspondence with the transition 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. The pattern $\beta(R)$ is used to define the bisimulation equivalence in the last axiom. $R \subseteq \beta(R)$ says that R is a post-fixpoint, i.e, a bisimulation. □

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

$$\mathsf{BPA} \models (R \subseteq R' \land R' \subseteq \beta(R')) \to (R \subseteq \approx_{\mathrm{BPA}}) \qquad (\textsc{CoindBPA})$$

where $R : PTerm \otimes PTerm$.

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

Having an ML specification for BPA, we may use ML reasoning to prove properties of process terms.

Proposition 8.2. *The following hold:*

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

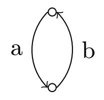
Explanation. We show the proof trees for Item 1 and leave the rest as exercises. For notational simplicity let us define $\Phi \equiv \exists p{:}PTerm.\, \langle p + p, p\rangle$.

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

□

Infinite processes are specified using guarded recursive specifications [17]. Here is a very simple example:

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



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

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

However, there is a subtle difference between the definition of the solution of a recursive specification in process algebra and that given by its ML encoding. In process algebra, a solution is given by any pair $\langle p_1, p_2 \rangle$ of processes such that $p_1 \approx a\,; p_1$ and $p_2 \approx b\,; p_2$. In the ML encoding, px and py designate sets of processes. Therefore, in order to capture the definition from process algebra, we have to prove that $\forall p, q.\, p \in px \wedge q \in px \to p \approx q$ (and similarly for py). For the above example, these properties follow from the fact that px and py are singleton: $\exists p_1.\, px = p_1$ and $\exists p_2.\, py = p_2$.

Having processes encoded in ML, we may use ML patterns to express their properties. For instance, we can show that the processes specified above include infinite loops: $px \vee py \to \nu Y.\, \bullet_{\{a,b\}}\, Y$, where $\bullet_U\, \varphi \equiv \bigvee_{u \in U} \bullet_u\, \varphi$.

The approach explained on the above example can be straightforwardly extended to processes specified by a system of n equations $X_i = t_i(X_1, \ldots, X_n)$, $i = 1, \ldots, n$. For instance, the above system is encoded by

$$\langle px_1, \ldots, px_n \rangle = \nu P{:}PTerm \otimes \cdots \otimes PTerm.\, t(\pi_1(P), \ldots, \pi_n(P))$$

Process algebra is extended with additional rules to obtain transition system semantics:

$$\frac{t_i(p_1, \ldots, p_n) \xrightarrow{u} \surd}{p_i \xrightarrow{u} \surd} \qquad \frac{t_i(p_1, \ldots, p_n) \xrightarrow{u} y}{p_i \xrightarrow{u} y}$$

where $\langle p_1, \ldots, p_n \rangle$ is the solution of the system. With ML encoding, the above rules become internal theorems proved using ML reasoning.

9 Defining Functors and (Co)Monads as Matching Logic Specifications

In this section we show how 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) :
$(id\ s){:}s \ominus\!\!\!\to s$
$\forall x{:}s.\, (id\ s)\, x = x$
$\forall g_1{:}s_1 \ominus\!\!\!\to s_2.\, \forall g_2{:}s_2 \ominus\!\!\!\to s_3.\, (g_2 \circ g_1){:}s_1 \ominus\!\!\!\to s_3$
$\forall x{:}s_1.\, \forall g_1{:}s_1 \ominus\!\!\!\to s_2.\, \forall g_2{:}s_2 \ominus\!\!\!\to s_3.\, (g_2 \circ g_1)\, x = g_2\, (g_1\, x)$
endspec

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

Proposition 9.1.

$$\begin{aligned}
&\mathit{CAT} \models \forall g{:}s_1 \ominus\!\!\!\!\rightarrow s_2.\,(g \circ (id\, s_1)) =^{s_1}_{ext} g \\
&\mathit{CAT} \models \forall g{:}s_1 \ominus\!\!\!\!\rightarrow s_2.\,((id\, s_2) \circ g) =^{s_1}_{ext} g && (\circ\textsc{Idl}) \\
&\mathit{CAT} \models \forall g_1{:}s_1 \ominus\!\!\!\!\rightarrow s_2\,.\forall g_2{:}s_2 \ominus\!\!\!\!\rightarrow s_3\,.\forall g_3{:}s_3 \ominus\!\!\!\!\rightarrow s_4\,.(g_3 \circ (g_2 \circ g_1)) =^{s_1}_{ext} ((g_3 \circ g_2) \circ g_1) && (\circ\textsc{Assoc})
\end{aligned}$$

The explanation is left as an exercise to the reader.

Remark 9.2. A reader familiar with the Haskell programming language already noticed that CAT can be seen as an ML axiomatization of Hask [?], the category of Haskell types and functions. An essential aspect fo this axiomatization is the use of extensional equality for functions.

Matching Logic specification of a functor It is given by means 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) :
$f : Sorts \to Sorts$
$\forall g{:}s_1 \ominus\!\!\!\!\rightarrow s_2.\,(map\, g){:}(f\, s_1) \ominus\!\!\!\!\rightarrow (f\, s_2)$
$(map\,(id\, s)) = id\, s$
$\forall h{:}s_1 \ominus\!\!\!\!\rightarrow s_2.\, \forall g{:}s_2 \ominus\!\!\!\!\rightarrow s_3.\, map\,(g \circ h) =^{m\, s_1}_{ext} (map\, g) \circ (map\, h)$ (MDist)
endspec

Explanation. The objects mapping is given by the first (Functor Laws) axiom and the arrows mapping is given by 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 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:

$$\begin{aligned}
\mu \circ (\eta\, m) &= \mathbf{1}_m \\
\mu \circ (m\, \eta) &= \mathbf{1}_m \\
\mu \circ (m\, \mu) &= \mu \circ (\mu m)
\end{aligned}$$

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 sort $s{:}Sorts$. Similarly, η associates an arrow $\eta_s : s \to m\, s$ for each $s{:}Sorts$. Note that the above equalities express the commutativity of diagrams in terms of category theory. In the ML encoding they become internal axioms.

Matching Logic Specification

spec MONAD
Import: FNCTR

Symbol: μ, η
Metavariable: element variables $s, s_1, s_2{:}\mathit{Sorts}$
Notation:
$\mu_s \equiv \mu\ s$
Axiom:
(Natural Transformations):
$\mu_s{:}((m \circ m)\ s) \bigcirc\!\!\!\!\!\rightarrow (m\, s)$
$\eta_s{:}s \bigcirc\!\!\!\!\!\rightarrow (m\, s)$
$\forall\, g{:}s_1 \bigcirc\!\!\!\!\!\rightarrow s_2.\, \eta_{s_2} \circ g =^{s_1}_{\mathit{ext}} (\mathit{map}\, g) \circ \eta_{s_1}$ (ηNt)
$\forall\, g{:}s_1 \bigcirc\!\!\!\!\!\rightarrow s_2.\, (\mu_{s_2} \circ (\mathit{map}\ \mathit{map}\ g)) =^{m\, m\, s_1}_{\mathit{ext}} (\mathit{map}\, g) \circ \mu_{s_1}$ (μNt)
(Diagram Commutativity)
$\mu_s \circ \eta_{(m\, s)} =^{s}_{\mathit{ext}} \mathit{id}_{(m\, s)}$ (ηIdR)
$\mu_s \circ (\mathit{map}\, \eta_s) =^{s}_{\mathit{ext}} \mathit{id}_{(m\, s)}$ (ηIdL)
$\mu_{m\, s} =^{m\, m\, m\, s}_{\mathit{ext}} \mathit{map}\, \mu_s$ $(\mu m\text{Comm})$
endspec

Explanation. We prefer to use the axiom $(\mu m\text{Comm})$ instead of (μAssoc) (see below), which is proved as semantic consequence. □

Proposition 9.3.

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

Explanation. (μAssoc) follows by applying $(\mu m\text{Comm})$. (μNt2) is explained as follows:

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

□

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 instead of μ a symbol *bind* (denoted also by `>>=`) defined by the following axiom, which we add to MONAD:

$$\forall\, g{:}s_1 \bigcirc\!\!\!\!\!\rightarrow m\, s_2.\, \mathit{bind}\, g =^{m\, s_1}_{\mathit{ext}} \mu_{s_2} \circ \mathit{map}\, g$$

It is easy to see that *bind* satisfies $\mathit{bind}{:}m\, s_1 \bigcirc\!\!\!\!\!\rightarrow (s_1 \bigcirc\!\!\!\!\!\rightarrow m\, s_2) \bigcirc\!\!\!\!\!\rightarrow m\, s_2$, which can be spelled out as follows:

$$\forall x{:}m\, s_1.\, \forall\, g{:}s_1 \bigcirc\!\!\!\!\!\rightarrow m\, s_2.\, \exists y{:}m\, s_2.\, \mathit{bind}\, g\, x = y.$$

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:

Proposition 9.4. *The following hold:*

1. $\mathit{MONAD} \models (\mathit{bind}\, g) \circ \eta_{s_1} =^{s_1}_{\mathit{ext}} g$
2. $\mathit{MONAD} \models \mathit{bind}\, \eta_s =^{m\, s}_{\mathit{ext}} \mathit{id}_{m\, s}$
3. $\mathit{MONAD} \models \forall\, g{:}s_1 \bigcirc\!\!\!\!\!\rightarrow m\, s_2.\, \forall\, h{:}s_2 \bigcirc\!\!\!\!\!\rightarrow m\, s_3.\, \mathit{bind}((\mathit{bind}\, h) \circ g) =^{m\, s_3}_{\mathit{ext}} \mathit{bind}\, h \circ \mathit{bind}\, g$

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

$$(\mathit{bind}\, g) \circ \eta_{s_1} =^{m\, s_1}_{\mathit{ext}} (\mu_{s_2} \circ (\mathit{map}\, g)) \circ \eta_{s_1} \qquad \text{(by definition of } \mathit{bind})$$

$$
\begin{array}{lr}
=^{m\,s_1}_{ext}\ \mu_{s_2}\circ((map\,g)\circ\eta_{s_1}) & \text{(by } (\circ\textsc{Assoc})) \\
=^{m\,s_1}_{ext}\ \mu_{s_2}\circ(\eta_{m\,s_2}\circ g) & \text{(by } (\eta\textsc{Nt}) \\
=^{m\,s_1}_{ext}\ (\mu_{s_2}\circ\eta_{m\,s_2})\circ g & \text{(by } (\circ\textsc{Assoc})) \\
=^{m\,s_1}_{ext}\ id_{m\,s_2}\circ g & \text{(by } (\eta\textsc{IdR})) \\
=^{m\,s_1}_{ext}\ g & \text{(by } (\circ\textsc{IdL}))
\end{array}
$$

(Item 2).

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

(Item 3).

$$
\begin{array}{lr}
bind((bind\,h)\circ g) & \\
\quad =^{m\,s_3}_{ext}\ bind((\mu_{s_3}\circ(map\,h))\circ g) & \text{(by definition of } bind) \\
\quad =^{m\,s_1}_{ext}\ \mu_{s_3}\circ map\,((\mu_{s_3}\circ(map\,h))\circ g) & \text{(by definition of } bind) \\
\quad =^{m\,s_1}_{ext}\ \mu_{s_3}\circ map\,(\mu_{s_3}\circ(map\,h))\circ(map\,g) & \text{(by (MDist),} (\circ\textsc{Assoc})) \\
\quad =^{m\,s_1}_{ext}\ \mu_{s_3}\circ((map\,h)\circ\mu_{s_2})\circ(map\,g) & \text{(by } (\mu\textsc{Nt2})) \\
\quad =^{m\,s_1}_{ext}\ (\mu_{s_3}\circ(map\,h))\circ(\mu_{s_2}\circ(map\,g)) & \text{(by } (\circ\textsc{Assoc})) \\
\quad =^{m\,s_1}_{ext}\ bind\,h\circ bind\,g & \text{(by definition of } bind)
\end{array}
$$

□

Remark 9.5. MONAD can be seen as a partial ML axiomatization of the monads in Haskell [20]. The definition of the Haskell class Monad is as follows:

```
class Monad m where
  (>>=)  :: m a -> (  a -> m b) -> m b
  (>>)   :: m a ->  m b         -> m b
  return ::   a                 -> m a
  fail   :: String -> m a
```

The correspondence between the members of the class `Monad` and those of MONAD is:

`(>>=) :: m a -> (a -> m b) -> m b`	$bind{:}m\,a \ominus (a \ominus m\,b) \ominus m\,b$
`return :: a -> m a`	$\eta_a{:}a \ominus m\,a$

In the Haskell documentation, it is written that the members of the class `Monad` should obey the following equations:

```
return a >>= k                  =  k a
m        >>= return             =  m
m        >>= (\x -> k x >>= h)  =  (m >>= k) >>= h
```

which are exactly the properties proved in Proposition 9.4. In order to get a full axiomatization for the Haskell monad, we need to add the symbols `>>` and `fail`, together with their axioms, to MONAD.

9.3 Comonads

9.3.1 Comonads Categorically

In category theory the notion of comonad is defined as the dual of that of monad. Consequently, a comonad 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 sort $s{:}Sorts$. Similarly, ε associates an arrow $\varepsilon_s : w\,s \to s$ for each $s{:}Sorts$. Note that the above equalities express the commutativity of diagrams in category theory terms.

Matching Logic Specification

spec COMONAD
  Import: FNCTR
  Symbol: δ, ε
  Notation:
    $\delta_s \equiv \delta\,s$
    $\varepsilon_s \equiv \varepsilon\,s$
  Axiom:
    (NATURAL TRANSFORMATIONS) :
      $\forall s{:}Sorts.\,\delta_s{:}(w\circ w\,s)\ominus(w\,s)$
      $\forall s{:}Sorts.\,\varepsilon_s{:}(w\,s)\ominus s$
      $\forall\,g{:}s_1\ominus s_2.\,g\circ\varepsilon_{s_1} =^{w\,s_1}_{ext} \varepsilon_{s_2}\circ(map\,g)$ (εNT)
      $\forall\,g{:}s_1\ominus s_2.\,\delta_{s_2}\circ(map\,g) =^{w\,w\,s_1}_{ext} (map\,map\,g)\circ\delta_{s_1}$ (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

Proposition 9.6.

$$COMONAD \models \forall s{:}Sorts.\,(map\,\delta_s)\circ\delta_s =^{w\,s}_{ext} \delta_{w\,s}\circ\delta_s \qquad (\delta\text{ASSOC})$$
$$COMONAD \models \forall\,g{:}w\,s_1\ominus s_2.\,map\,((map\,g)\circ\delta_{s_1}) =^{w\,w\,s_1}_{ext} \delta_{s_2}\circ(map\,g) \qquad (\delta\text{NT2})$$

Explanation. Similar to that of Proposition 9.3. □

9.3.2 Comonads in Functional Programming Languages

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

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

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

Proposition 9.7. *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* $\models \forall\,g{:}w\,s_1\ominus s_2.\forall\,h{:}w\,s_2\ominus s_3.\,cobind(h\circ(cobind\,g)) =^{m\,s_1}_{ext} cobind\,h\circ cobind\,g$

Explanation. Similar to that of Proposition 9.4. □

Remark 9.8. The Haskell class `Comonad` is defined as follows [21]:

```
class Functor w => Comonad w where
    extract :: w a -> a
    duplicate :: w a -> w (w a)
    extend :: (w a -> b) -> w a -> w b
```

The correspondence between this class and COMONAD is almost obvious:

`extract :: w a -> a`	$\varepsilon_a{:}w\,a \mathrel{\ominus\!\!\!\rightarrow} a$
`duplicate :: w a -> w (w a)`	$\delta_a{:}w\,a \mathrel{\ominus\!\!\!\rightarrow} w\,w\,a$
`extend :: (w a -> b) -> w a -> w b`	$\mathit{cobind}{:}(\,w\,a \mathrel{\ominus\!\!\!\rightarrow} b\,) \mathrel{\ominus\!\!\!\rightarrow} w\,a \mathrel{\ominus\!\!\!\rightarrow} w\,b$

The equations the class should obey

```
extend extract      = id
extract . extend f  = f
extend f . extend g = extend (f . extend g)
```

are exactly those given by Proposition 9.7.

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.

Matching logic follows a minimalism design. Matching logic syntax has only 8 syntactic constructs that are the most basic ones such as variables, (user-provided) symbols, application, propositional connectives, quantification, and fixpoints, leaving the more complex concepts such as equality, functions, predicates, sorts, many-sorted and/or order-sorted structures, inductive and coinductive data structures, types, and so on as definable concepts. In the paper, we showed how to define these concepts as matching logic specifications.

We focused on the expressiveness of matching logic by presenting matching logic specifications that define a variety of logical systems, particularly those that have direct support for induction and/or coinduction and for dealing with data types. From these specifications, we demonstrated the flexibility of matching logic.

The minimalism design of matching logic also includes the fact that it has a relatively small proof system, as shown in Section 5. As a result, it is relatively simple to implement an algorithm that checks the correctness of a matching logic Hilbert-style proof. Such an algorithm is called the *proof checking* algorithm. Therefore, matching logic is preferred if one cares about a small trusted core for all formal reasoning.

Matching logic is particularly useful in defining the formal semantics of programming language. For example, the $\mathbb{K}$ formal language framework (`http://kframework.org`) uses matching logic as its logical foundation. In $\mathbb{K}$, the syntax of programming languages is declared using BNF grammars, which translates into inductive data types built from the constructors (which are language constructs) in matching logic. The language semantics is defined in terms of rewrite rules of the form $\varphi_1 \Rightarrow \varphi_2$, where φ_1 and φ_2 are matching logic patterns that can be matched by the computation configurations of the languages. In other words, the formal definition of a programming language yields a corresponding logical theory in matching logic. We refer the reader to [?, ?] for more discussion.

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, A general approach to define binders using matching logic, Tech. Rep. `http://hdl.handle.net/2142/106608`, University of Illinois at Urbana-Champaign (2020).

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

[5] 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`.

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

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

[8] 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).

[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`

[21] Haskell monads, last visit December 2019.
URL `http://hackage.haskell.org/package/comonad`